MASTER EN SCIENCES INFORMATIQUES

Élaboration d'un algorithme efficace de recherche de cliques

Badoux, Florence

Award date:
1993

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Rue Grandgagnage, 21
B-5000 Namur (Belgium)

Elaboration d'un algorithme efficace de recherche de cliques

Florence BADOUX

Promoteurs: J. Fichefet
E. DEPIEREUX

Mémoire présenté en vue de l'obtention du diplôme de Licencié et Maître en Informatique

Année Académique 1992- 1993

Je remercie Messieurs Fichefet et Depiereux de m'avoir permis de réaliser ce mémoire et d'avoir supervisé celui-ci.

Je tiens à remercier également toutes les personnes qui m'ont entourée, encouragée et soutenue tout au long de ce mémoire; et plus spécialement je tiens à remercier Jacques pour la patience qu'il a eue, l'aide qu'il m'a apportée, Marcel Rémond pour ses précieux conseils, Philippe, Myriam, Ariane, Agnès et Catherine pour l'aide apportée à la rédaction de ce mémoire.

Résumé

Mots-clés : Clique, algorithme efficace, problème NP-complet.

Ce mémoire concerne la recherche d'un algorithme efficace pour résoudre un problème rencontré dans une méthode d'alignement de séquences de protéines. Cette méthode a été développée par l'unité de biologie quantitative des Facultés Notre-Dame de la Paix. Ce problème correspond en réalité à un problème d'optimisation combinatoire bien connu en théorie des graphes :

"La recherche de toutes les cliques maximums dans un graphe G = (V,E)".

Une clique dans un graphe G est un sous-graphe complet maximal.

Ce problème étant NP-complet, l'existence d'un algorithme polynomial pour le résoudre semble improbable. Les algorithmes généralement trouvés dans la littérature solutionne le problème en un temps raisonnable mais exponentiel. Ils se basent sur des méthodes de recherche avec rebroussement et des méthodes "branch-and-bound".

Nous proposons, ici un algorithme, basé également sur ces techniques, mais tout à fait original de par la condition limite ("bound condition") qu'il utilise. Notre problème est en fait un cas particulier de la recherche de toutes les cliques maximums: d'une part, la taille des cliques maximums est connue et d'autre part, nous travaillons en population groupée. Il existe des groupes de sommets dans le graphe, et une clique ne peut être formée que des sommets appartenants à des groupes différents. Et les clique recherchées doivent absolument faire intervenir un sommet de chaque groupe. Nous recherchons donc les cliques d'une taille donnée maximum, égale au nombre de groupes de sommets. Il s'avère que la comparaison des temps d'exécution des différents algorithmes montrent un avantage pour l'algorithme que nous avons conçu.

Abstract

Key words : Clique, efficient algorithm, NP-complete problem.

The purpose of this thesis is to find an efficient algorithm able to deal with a problem raised by a simultaneous alignment of several protein sequences.

This method has been developed by the Quantitative Biology Unit of the "Facultés Notre-Dame de la Paix". This problem consist in combinatory optimisation problem well known in graph theory.

The search for all maximum cliques in G=(V,E) graph.

A clique in a G graph is a complete maximal NP-graph. This problem being NP-complete, the existence of a polynomial algorithm to solve it seems unprobable.

In the literature, the generally found algorithms solve the problem in reasonable but exponential time.

These algorithms generally used the "backtracking" and "branch-and-bound" techniques. We develop an algorithm which is also based on these techniques but which is innovative because it used a totally new bound condition.

Our problem is in fact a particular case of the usual maximum clique problem. On one hand, the size of the maximum cliques is known and on the other hand, we deal with a aggregated population. There exists vertices groups in the graph and a clique can only be built with vertices belonging to different groups. Moreover, the searched clique must be concerned by a vertex having a maximum given size which is equal to the number of vertices groups. With the comparison of the running time of different algorithms, we show that our algorithm dominates others.

I. Introduction

Ce mémoire s'intègre dans le cadre d'une recherche en biologie moléculaire, réalisée par l'unité de biologie quantitative des Facultés Universitaires Notre-Dame de la Paix. E. Depiereux et E. Feytmans ont développé une méthode d'alignement de séquences de protéines qui a pour objectif de prédire les régions structurellement conservées au sein des protéines. Elle met en oeuvre un ensemble de procédures informatiques qui permettent d'aligner plusieurs séquences de protéines. Cependant cette méthode possède certaines limites qui seront soulevées au point II.C. Une des procédure, le Screening, est confrontée à un problème : la détection de tous les "matchs" complets. Un "match" (appariement), est une correspondance définie entre deux segments de taille identique issus de deux séquences de protéines différentes si le degré de similarité entre eux est plus élevé que le degré que l'on obtiendrait par hasard.

Un "match" complet est un ensemble de segments de longueur égale dans lequel :

1. chaque segment appartient à une séquence différente,
2. le nombre de segments formant le "match" complet est égal au nombre de séquences,
3. tous les segments sélectionnés sont similaires entre eux deux à deux.

Historiquement, il a été montré qu'il existait une corrélation entre ce problème et un problème de théorie des graphes bien connu : la recherche des cliques maximums. L'unité de biologie quantitative disposait d'une "solution" informatique à ce problème, un programme de génération de cliques écrit par Judy Hempel. Toutefois cette solution était trop lente pour pouvoir être intégrée dans l'ensemble des procédures de la méthode d'alignement. Il fallait donc l'optimiser ou trouver une nouvelle solution.

Après l'étude du programme de Judy Hempel, il s'est avéré qu'il serait difficile de modifier ce programme en vue de son amélioration. Étant donné que nous disposions de peu d'information à propos du problème de cliques, nous nous sommes orientés, dans un premier temps, vers la recherche d'une solution dans la littérature.

Les articles concernant le problème des cliques font intervenir de nombreuses notations que nous avons expliquées. Ensuite, nous avons énoncé le problème des cliques proprement dit.

En reprenant les idées intéressantes de la littérature et en y ajoutant quelques caractéristiques propres au problème biologique étudié, nous nous sommes dirigés vers la réalisation de notre propre algorithme.

II. Problème biologique

Ce mémoire trouve son point de départ dans l'existence d'un problème de méthode d'alignement de séquence de protéine. Avant de l'expliquer, nous présentons quelques bases de biologie moléculaire, principalement à l'attention des informaticiens.

A. Notions fondamentales de biologie moléculaire

Les protéines sont des macromolécules qui ont un rôle crucial dans pratiquement tous les processus vitaux. Elles ont une très grande importance et une activité étendue dans les processus biologiques tels que ceux décrits dans les lignes qui suivent.

— La catalyse enzymatique. Dans les systèmes biologiques, pratiquement toutes les réactions chimiques sont catalysées par des protéines spécifiques, appelées enzymes. Elles augmentent la vitesse des réactions.
— Le transport et le stockage. Beaucoup de petites molécules et d'ions sont transportés par des protéines spécifiques, telle que l'hémoglobine, par exemple, qui transporte l'oxygène dans les globules rouges.
— Les mouvements coordonnés. Les protéines sont les composantes majeures des muscles.
— Le support mécanique, réalisé par exemple, par le collagène des os et du cartilage.
— La protection immunitaire. Les anticorps sont des protéines très spécifiques qui peuvent reconnaître des substances étrangères;
— ...

Les protéines sont construites sur base de vingt acides aminés différents (voir annexe 1, fig. II.1). Ces acides sont reliés les uns aux autres par des liaisons peptidiques et forment une chaîne non ramifiée. Cette chaîne nommée peptide lorsqu'elle est constituée d'un petit nombre d'acides aminés, et polypeptide lorsqu'elle est formée par un enchaînement plus long. Une protéine est formée d'une ou plusieurs chaînes polypeptidiques.

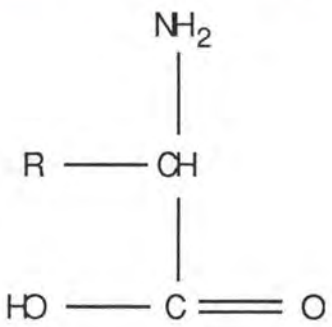


Fig. II.2

Les acides aminés sont de petites molécules formées d'un squelette commun, d'une chaîne principale, et d'une chaîne latérale (fig. II.2). Cette chaîne latérale permet de les distinguer et de déterminer leurs propriétés structurales et physico-chimiques, telles que l'hydrophobicité, leur charge, leur taille, leur forme, leur capacité à former des ponts hydrogènes, et leur réactivité chimique.

L'origine de cet alphabet fondamental des protéines remonte à au moins deux milliards d'années.

La gamme remarquablement étendue des fonctions médiées par les protéines, provient de la diversité des acides aminés et de leur séquence au sein des protéines. Cette séquence confère aux protéines une structure spatiale tridimensionnelle. C'est cette structure tridimensionnelle qui détermine leur fonction biologique.

Généralement, on distingue quatre niveaux de structuration dans l'architecture des protéines :

— la structure primaire, qui correspond à la séquence en acides aminés (ou résidus);
— la structure secondaire, qui fait référence à l'arrangement spatial des acides aminés, ou résidus, ainsi qu'aux interactions entre acides aminés au niveau de leur chaîne principale; certaines de ces relations stériques possèdent une régularité donnant lieu à une structure périodique particulière. On distingue généralement trois sortes de structures secondaires : l'hélice α (voir annexe 1 fig. II.3), le feuillet β (voir annexe 1 fig. II.4) et le "turn" (le virage) (voir annexe 1 fig. II.5);
— la structure tertiaire est l'arrangement spatial des acides aminés éloignés dans la séquence linéaire. Elle correspond à l'assemblage de structures secondaires en une structure appelée monomère;
— la structure quaternaire est un niveau de structuration présent pour les protéines comportant plusieurs chaînes polypeptidiques. Elle correspond à l'assemblage de structures tertiaires, monomères, en un polymère.

Comme nous l'avons déjà dit, la succession des acides aminés au sein des protéines confère à celles-ci une structure spatiale précise qui détermine leur fonction. Il est donc essentiel de pouvoir disposer de modèles tridimensionnels de protéines pour comprendre leur fonction biologique.

Grâce aux techniques de génétique moléculaire, le nombre de séquences en acides aminés de protéines qui sont déterminées augmentent de manière quasi exponentielle.

Néanmoins, la manière dont ces chaînes se replient dans l'espace est un problème non élucidé qui reste un des principaux défis intellectuels de la biologie moléculaire.

Comme la structure des protéines ne peut être prédite directement à partir de leur séquence, on la détermine expérimentalement par des techniques telle que la cristallographie aux rayons-X ou la technique de résonance magnétique nucléaire (NMR). Durant les trente dernières années, la technique de cristallographie aux rayons-X a permis de déterminer expérimentalement la structure d'environ cinq cent protéines (Branden C. et Tooze J., 1991); ce qui est largement inférieur au nombre de séquences en acides aminés de protéines disponibles (environ 100.000).

D'autre part, l'apparition d'ordinateurs graphiques puissants et d'autre part, les logiciels associés, ont permis la manipulation des structures tridimensionnelles déterminées aux rayons-X, ainsi que la construction de modèles tridimensionnels de protéines de structure inconnue, sur base de protéines homologues de structure connue.

Le remplacement spécifique de résidus dans la séquence de protéines, ou aussi mutagenèse dirigée; en génétique moléculaire, permet l'ingénierie de nouvelles protéines possédant une spécificité élargie pour le substrat, une meilleure stabilité thermique, ou parfois même une nouvelle fonction (Clarke, 1989). L'alignement des séquences de protéines de structure connue avec les séquences de protéines de structure inconnue permet de mettre en évidence les résidus conservés, et donc probablement importants fonctionnellement ou structurellement, au sein d'une famille de protéines par exemple. Les alignements permettent aussi de déterminer des zones possédant des caractéristiques physico-chimiques semblables dans toutes les séquences alignées, et probablement conservées au niveau structural.

B. Méthode d'alignement multiple d'E. Depiereux et d'E. Feytmans

E. Depiereux et E. Feytmans ont proposé une méthode d'alignement originale, la méthode Match-Box, pour aligner plusieurs séquences de protéines simultanément (Depiereux E. et Feymans E., 1991). Cette méthode permet de détecter des régions similaires entre les différentes séquences de protéines alignées, et de prédire ainsi les régions qui seraient structurellement et/ou fonctionnellement conservées. La méthode d'alignement repose sur deux principes de base :

1. chaque acide aminé est caractérisé par un **profil physico-chimique multivarié**, c'est-à-dire que les différentes propriétés physico-chimiques des acides aminés sont prises en compte pour détecter les similarités;
2. toutes les séquences à aligner sont considérées **simultanément** et l'algorithme détecte des ensembles de courts segments qui présentent un profil physico-chimique similaire.

Bases théoriques et méthodologie générale :

Considérons une comparaison entre un ensemble de r séquences de protéines (de 200 à 300 acides aminés chacune), généralement de longueur l_i, différente, avec $i = 1, \ldots, r$.

L'alignement simultané et la prédiction des régions conservées sont réalisés par trois approches complémentaires :

1. le "SCANNING" de chaque paire de séquences;
2. le "MATCHING" de segments ayant des profils physico-chimiques similaires, selon un degré de signification statistique;
3. le "SCREENING" des **"matchs"** (correspondances) **complets** produisant le meilleur alignement.

1. "Scanning" des séquences

Une **fenêtre**est définie comme étant n'importe quel segment de longueur constante *w* (nombre d'acides aminés). La longueur de cette fenêtre est déterminée par l'utilisateur et reste constante tout au long de la procédure.

Un **"scan"**, une lecture, est réalisée pour chaque séquence par deux boucles imbriquées :

- une **fenêtre initiale** est déplacée progressivement, par étapes d'un seul acide aminé, du premier au dernier résidu de la séquence *i*, à chaque position de la fenêtre initiale (voir fig. II.6);

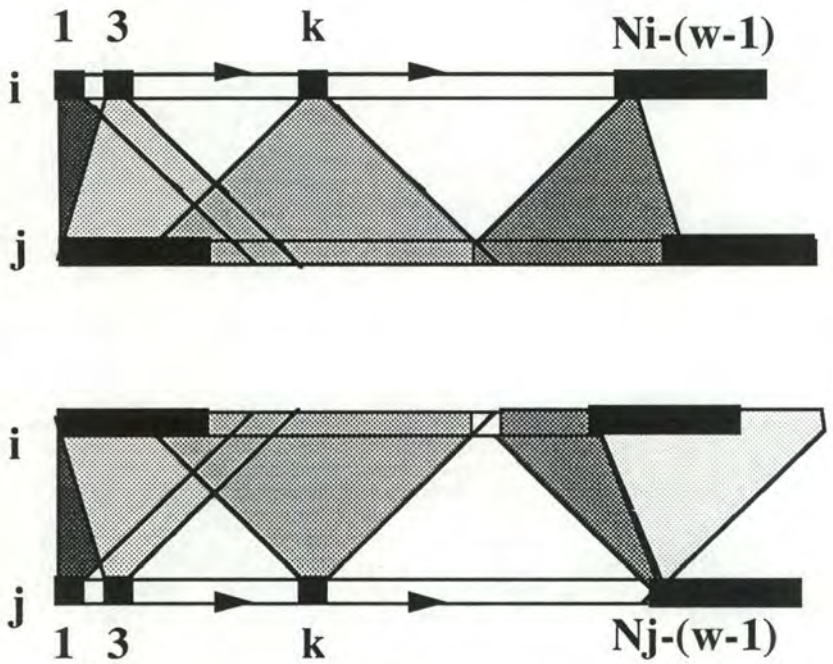


Fig II.6

- une **fenêtre mobile** est déplacée le long de toutes les autres séquences, par étapes d'un acide aminé, avec un déplacement positif ou négatif de *m* acides aminés de part et d'autre de la position de la fenêtre initiale (voir fig. II.7). A chaque étape, le segment défini par la fenêtre mobile est comparé au segment défini par la fenêtre initiale selon la procédure de "matching", correspondance, décrite ci-dessous.

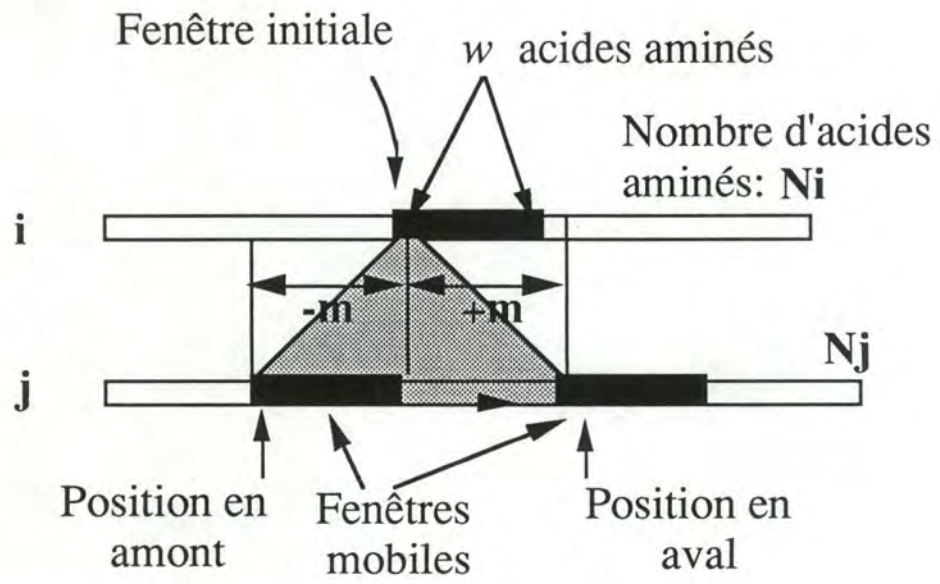


Fig II.7

Il faut noter que pour une paire de séquence i,j le processus implique que la fenêtre mobile sera déplacée dans j pour chaque position de la fen$être initiale de i, et plus tard, elle sera déplacée dans i pour chaque position de la fenêtre initiale de j. Les comparaisons réciproques ne fournissent pas nécessairement des résultats redondants.

2. "Matching" de deux segments

Considérons la comparaison entre deux segments de taille identique (voir fig. II.8).

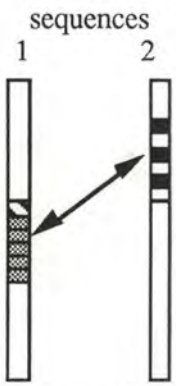


Fig. II.8

A moins que tous les acides aminés d'un segment ne soient identiques aux acides aminés correspondants de l'autre segment, la détermination quantitative du degré de similarité entre les segments n'est pas triviale. Deux critères essentiels peuvent être envisagés :

1. le **nombre d'identités** entre les acides aminés correspondants est plus grand ou égal à la plus grande valeur attendue pour des séquences générées de manière aléatoire;

2. une **mesure de similarité** calculée à partir du profil physico-chimiques des acides aminés est plus grande que la plus grande valeur attendue pour des séquences générées de manière aléatoire :

$$D2 = \sum_{i=1}^{w} \sum_{j=1}^{n} \frac{(Zij1 - Zij2)^2}{2}$$

A chaque étape de la procédure de "scanning", les segments correspondants à la fenêtre initiale et à la fenêtre mobile sont comparés selon une combinaisone des deux critères décrits ci-dessus.

Un **"match" (appariement)**, une correspondance, est définie entre deux segments si le degré de similarité entre eux est plus élevé que ce qu'on attendrait par hasard. A chaque fenêtre initiale, peuvent être associées un ou plusieurs "matchs" dans une ou plusieurs séquences. Le nombre de "matchs" observés dépend de la similarité entre les régions analysées.

3. "Screening" des "matchs" complets.

Considérons un groupe de *r* séquences. Après les procédures de "scanning" et de "matching", la mesure de similarité n'étant pas transitive, il reste à déterminer si les segments sélectionnés dans chacune des séquences forment un groupe complet.

Considérons les séquences en acides aminés de trois segments de protéines suivants et prenons comme critère de similarité, par exemple que : "deux segments de protéines sont similaires s'ils ont au moins 4 acides aminés identiques".

A	B	C
G	G	I
G	G	I
G	G	I
G	G	I
A	S	S
A	S	S
A	S	S
A	S	S

Prenons comme critère que deux segments de protéines sont similaires s'ils ont au moins quatre acides aminés identiques. Soit A, B, C trois segments pris dans trois séquences de protéines différentes. On considérera que A et B sont similaires, B et C aussi mais pas A et C. Par conséquent, comme la mesure de similarité n'est pas transitive, deux définitions peuvent être proposées pour définir un **groupe :**

1. A et B et C forment un groupe si et seulement si A "match" avec B et C ou si B "match" avec A et C, ou si C "match" avec A et B; on appelle cela un **lien simple.**
2. A et B et C forment un groupe ssi A "match" avec B et C et B "match" avec C; on appelle cela un **lien complet.**

Ce principe peut être étendu à plus de trois segments. Un segment D peut être ajouté au groupe s'il s'apparie au moins avec un des membres du groupe, dans le cas d'un lien simple; ou s'il s'apparie avec tous les membres du groupe, dans le cas d'un lien complet. Si l'on considère le lien simple, l'effet de chaîne (A s'apparie avec B, C et D, B s'apparie avec C, D s'apparie avec A) peut conduire à ce que D soit très différent de B.

Plusieurs "matchs" peuvent être trouvés à l'intérieur d'une même séquence. Mais si l'on désire aligner les séquences pour faire correspondre les zones similaires, un "match" au plus doit être considéré dans l'alignement (si l'on présume qu'il n'existe pas de duplication). Dans ce qui suit, nous considérerons que des segments de longueur égale forment un **"match" complet** si :

1. chaque segment appartient à une séquence différente,
2. le nombre de segments formant le "match" complet est égal au nombre de séquences,
3. et le groupe des segments sélectionnés forme un **lien complet** selon le critère considéré dans la procédure de "matching".

Les fenêtres initiales peuvent être similaires à plusieurs fenêtres mobiles et donner, par conséquent, plusieurs "matchs" pouvant même appartenir à une même séquence.

Le nombre de "matchs" complets potentiels augmente de façon exponentielle avec le nombre de séquences et avec le nombre de "matchs" trouvés par séquence. L'exemple de la figure II.9 l'exprime clairement.

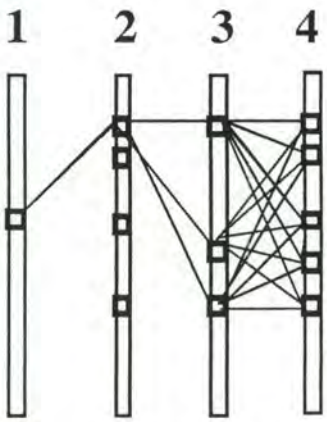


Fig II.9

Pour sélectionner des "matchs" complets, la méthode actuellement utilisée est **la méthode de la distance-minimum**. Cette méthode recherche les "matchs" complets en deux étapes.

Premièrement, elle sélectionne pour chaque fenêtre initiale, le "meilleur match" dans chacune des autres. Si plusieurs matchs satisfont les critères statistiques énoncés plus haut, cette sélection est faite selon les critères suivants, par ordre décroissant de priorité :

1. le plus grand nombre d'identités,
2. la plus petite valeur D^2 (distance carrée Euclidienne) entre les profils physico-chimiques,
3. la plus petite différence entre les positions des fenêtres dans les deux séquences.

Deuxièmement, un test est réalisé pour voir si cette sélection forme un "match" complet.

En résumé, pour un segment de longueur W situé à une position donnée de la séquence i (fenêtre initiale), tous les segments de chacune des autres séquences, situés entre des limites fixées par un paramètre m, sont passés en revue. Les segments qui satisfont à un test de similarité sont retenus, et forment un ensemble de "matchs" dans chacune des séquences. Les combinaisons qui comprennent la fenêtre initiale et un match par séquence sont tellement nombreuses que le problème est actuellement simplifié en sélectionnant un seul match pas séquence, l'information potentielle des autres segments étant définitivement perdue. Si l'ensemble des segments sélectionnés forme un match complet, celui-ci est stocké en vue de la procédure du screening. Dans le cas contraire, aucune information n'est retenue pour cette fenêtre initiale.

C. Problème sous-jacent à la méthode

Pour réaliser l'alignement final des séquences des protéines, il faut mettre en correspondance les régions communes au niveau de leur similarité; ce que nous faisons en introduisant des interruptions dans la séquence (voir Fig. II.10). Pour cela, il est souvent nécessaire de faire des choix pour déterminer quels sont les "matchs" complets significatifs.

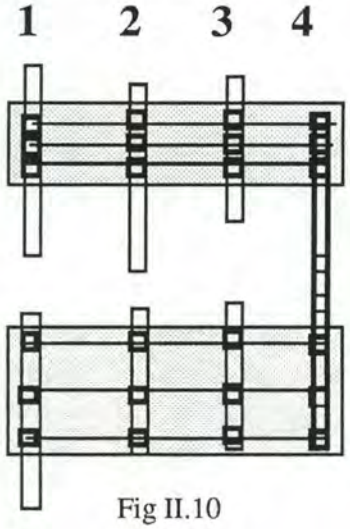


Fig II.10

Les "matchs" complets sont considérés comme significatifs s'ils forment le plus grand ensemble compatible avec le même schéma d'interruptions. Tandis que les "matchs" complets qui chevauchent un schéma d'interruptions significatives, sont considérés comme du bruit de fond (voir fig. II.11). L'alignement final est réalisé en sélectionnant les "matchs" complets significatifs parmi tous les "matchs" complets trouvés à partir de toutes les fenêtres initiales.

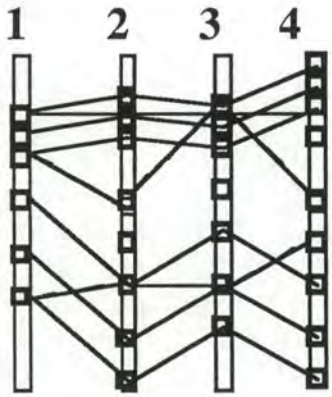


Fig II.11

Des régions similaires dans plusieurs séquences, non interrompues, peuvent être délimitées par une boîte. L'alignement de plusieurs boîtes nécessite des interruptions qui correspondent à des résidus (acides aminés) trouvés dans certaines séquences et non dans d'autres. L'alignement optimal correspond à l'ensemble le plus long de boîte qui peuvent être simultanément alignés sans interrompre l'ordonnancement des résidus dans les séquences (voir fig. II.12).

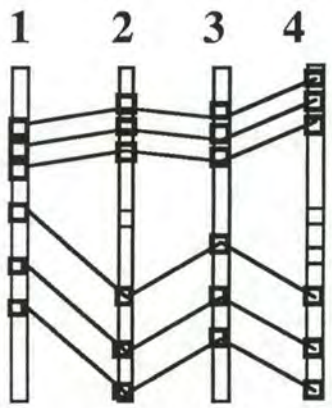


Fig II.12

De nombreux tests de performance de la méthode de la distance-minimum ont été réalisés par des alignements de séquences de protéines de structure connue. Ces tests mettent en évidence que les régions où les protéines étaient les plus similaires, du point de vue séquence (distance de similarité minimum), coïncidaient aux régions structurales conservées dans 77% des cas. Mais dans 23% des cas, alors que l'alignement des séquences aurait prédit certaines régions comme structurellement conservées, il s'avère que des "matchs" complets, non sélectionnés par la méthode de la distance-minimum, auraient donné un meilleur alignement structural. La méthode d'alignement, avec la stratégie de la distance-minimum, échoue parfois

dans la prédiction des régions structurellement conservées. Il n'existe donc aucun argument *a priori* qui permette de choisir directement les "matchs" complets qui correspondraient à un alignement optimal.

Il est donc nécessaire de rechercher une nouvelle stratégie dans la procédure de "screening". Cette nouvelle procédure permet d'inclure tous les "matchs" obtenus, pour une limite de similarité donnée, afin de trouver tous les "matchs" complets.

Comme nous l'avons dit précédemment, le nombre de "matchs" complets potentiels augmente de façon très importante avec le nombre de séquences et de "matchs" trouvés par séquence. Si, par exemple, d'une part, nous considérons un alignement de 6 séquences, avec environ 10 "matchs" par séquence pour une seule fenêtre initiale, et d'autre part, qu'il y a environ plusieurs milliers de fenêtres initiales à traiter dans une seule étape d'alignement, le problème devient extrêmement complexe. Jusqu'à présent, aucune solution algorithmique n'a été trouvée pour résoudre ce problème en un temps CPU satisfaisant.

Le problème biologique auquel nous sommes confrontés est de trouver une autre méthode, que celle de la distance-minimum, qui permettrait de trouver ***rapidement*** *le plus grand nombre possible de "matchs" complets.*

Une étude plus approfondie réalisée dans le cadre du développement de la méthode d'alignement a mis en évidence qu'il existait une corrélation entre ce problème et un problème de la théorie des graphes :

Le problème de clique dans un graphe.

Pour expliquer en quoi consiste exactement le problème de clique, nous allons tout d'abord donner quelques notions de théories des graphes. Nous formulerons ensuite, de manière précise, le problème de clique proprement dit. Nous en donnerons ensuite quelques applications générales et enfin, nous expliquerons la correspondance entre ce problème de théorie des graphes et notre problème biologique.

III. Problème des cliques en théorie des graphes

A. Notions élémentaires

1. Définitions

Nous allons donner ici quelques définitions des termes de la théorie des graphes que nous utiliserons. Il n'existe pas de terminologie standard. Celle que nous présentons ici est définie dans (56).

- Un **graphe** G = (V, E) consiste en un ensemble fini non vide V de **noeuds** ou **sommets** ("vertices") et un ensemble fini E d'**arêtes** ("edges"). Une arête est représentée par un couple de sommets appelés ses **extrémités**. Une arête est dite **incidente**.à ses extrémités. Le nombre de sommets du graphe G est appelé l'**ordre** de G.
- Un sous-graphe G' = (V',E') d'un graphe G = (V,E) est un graphe pour lequel $V' \subseteq V$ et $E' \subseteq E$.
- Deux sommets joints par une arête sont appelés **sommets adjacents** ou **voisins**. L'**application** $\Gamma(W)$, pour un ensemble de sommets W, donne l'ensemble des sommets adjacents aux sommets de W. $\Gamma(\{v\})$ s'écrit Γv. L'ensemble des arêtes dont une extrémité est le sommet $v \in V$ est noté $\delta(v)$. Le nombre $|\delta(v)|$ est le **degré**du sommet $v \in V$.
- Un **graphe** est **complet**si tous les sommets pris deux à deux sont joints par une arête; autrement dit, si tous les sommets sont adjacents deux à deux.
- Le **graphe complémentaire** $\overline{G}$ d'un graphe G, est un graphe qui a le même ensemble de sommets et dans lequel deux sommets sont adjacents si et seulement si ils ne sont pas adjacents dans G.
- Une **clique** dans un graphe G = (V, E) est un sous-ensemble de sommets de V où tous les sommets sont adjacents deux à deux.
- Un **ensemble de sommets indpendant** ("**stable set**" or "**independent set**") est un ensemble de sommets où tous les sommets pris deux à deux ne sont pas adjacents.

Comme nous l'avons dit, la terminologie en théorie des graphes n'est pas normalisée, d'autres définitions utilisées couramment dans la littérature et plus ou moins équivalentes sont les suivantes :

- G' est une **clique**du graphe G si G' est un sous-graphe complet [maximal].
  Pour certains auteurs, une clique, donc en plus de posséder la propriété d'être un graphe complet est aussi maximale.en ce sens qu'elle n'est inclue dans aucune autre. D'autres

font la distinction entre clique et clique maximale. Une clique est dans un graphe G est un ensemble de sommets indépendant [maximal] dans le graphe complémentaire $\overline{G}$.

- Un **tour**, rappelons-le, est une séquence finie, $T = v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$ où $(k \geq 0)$, commençant et terminant par un sommet, dans laquelle le sommet v_i et les arêtes e_j apparaissent alternativement, tel que pour $i = 1, 2, \ldots, k$, les extrémités de chaque arête e_i sont les sommets v_{i-1}, v_i.
- Les sommets v_0 et v_k sont appelés **origine** et **extrémité terminale**, ou **extrémités** de T. Les sommets $v_1, \ldots, v_{k-1}$ sont appelés **sommets internes** du tour T. Le nombre **k** est la **longueur** d'un tour.
- Un tour dans lequel tous les sommets (arêtes) sont distincts est appelé un **chemin.**
- Si un sommet **s** est l'origine d'un tour T et **t** l'extrémité terminale de T, alors T est appelé un **(s, t)-tour.**
- Si un sommet **s** est l'origine d'un chemin C et **t** l'extrémité terminale de C, alors C est appelé un **(s, t)-chemin.**
- Deux sommets s, t d'un graphe G sont dits **connectés** si G contient un (s, t)-chemin.
- **G** est appelé **connecté** si tous les sommets de G sont connectés deux à deux.
- Les **composantes** d'un graphe G sont les sous-graphes connectés maximums du graphe G. Si W est un ensemble de sommets dans G = (V, E) alors **G-W** correspond au graphe obtenu en enlevant W, c'est-à-dire que l'ensemble de sommets de G-W est V\W et G-W contient toutes les arêtes de G qui ne sont pas incidentes à un sommet dans W. Par **G(W)** on note le **sous-graphe de G induit** par un ensemble de sommet $W \subset V$ de sommets.

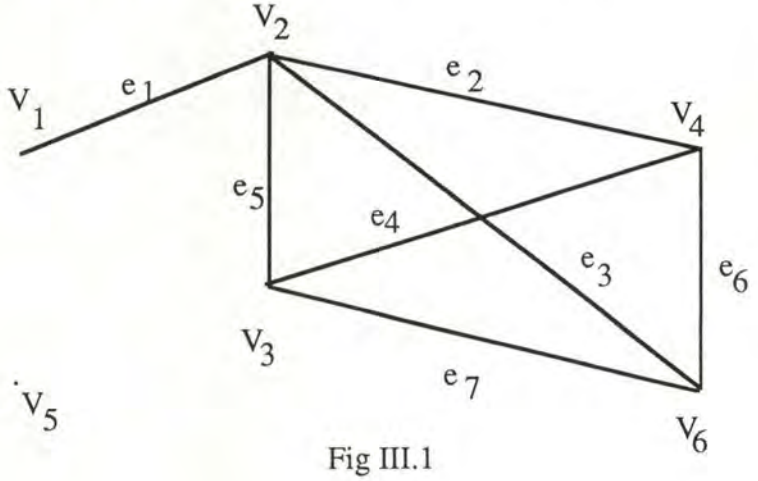
2. Illustration des définitions

Pour une meilleure familiarisation avec ces concepts et ces notations, nous vous proposons quelques exemples concrets.

Soit G = (V, E) le **graphe**, repris à la figure III.1. avec

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, l'ensemble des **sommets**

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, l'ensemble des **arêtes**



$e_1 = (v_1, v_2)$
$e_2 = (v_2, v_4)$
$e_3 = (v_2, v_6)$
$e_4 = (v_3, v_4)$
$e_5 = (v_2, v_3)$
$e_6 = (v_4, v_6)$
$e_7 = (v_3, v_6)$

Fig III.1

L'arête e_1 est **incidente** à v1 et v_2

L'arête e_2 est **incidente** à v_2 et v_4

L'arête e_3 est **incidente** à v_2 et v_6

L'arête e_4 est **incidente** à v_3 et v_4

L'arête e_5 est **incidente** à v_2 et v_3

L'arête e_6 est **incidente** à v_4 et v_6

L'arête e_7 est **incidente** à v_3 et v_6

L'ordre de G = 6.

v_1 et v_2, par exemple, sont **deux sommets adjacents**, v_2 et v_4 également.

Soient $W = \{v_1, v_3\}$

$\Gamma(W) = \{v_2, v_4, v_6\}$

$\Gamma(\{v_4\}) = \Gamma v_4 = \{v_2, v_3, v_6\}$,

$\delta(v_4) = \{e_2, e_4, e_7\}$ et $|\delta(v_4)| = 3$ = le **degré** de v_4 dans V, G n'est pas un **graphe complet**,

$\overline{G} = (V, E')$, il est repris à la figure III.2.

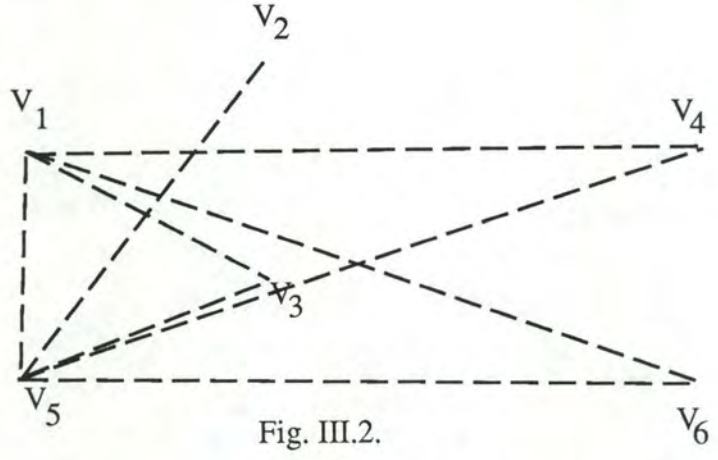


Fig. III.2.

Une **clique** de G est le sous-ensemble de sommets $\{v_2, v_4, v_3, v_6\}$, si l'on considère qu'une clique doit être un **graphe complet maximal**. Par contre, si l'on n'inclut pas la propriété "maximal", on aura les cliques $\{v_2, v_3\}$, $\{v_2, v_3, v_4\}$, ...

Un **ensemble de sommets indépendant** dans G, comme $\{v_1, v_3\}$, est une clique de $\overline{G}$.

Un **ensemble de sommets indépendant** dans $\overline{G}$, comme$\{v_2, v_4, v_3, v_6\}$, est une clique de G.

La séquence $T = v_1, e_1, v_2, e_3, v_6, e_7, v_4, e_2, v_2$ est un **tour** de G, où
- v_1 et v_2 sont l'origine et l'extrémité terminale de T, ou simplement les **extrémités** de T
- v_2, v_6, v_4 sont les **sommets** internes de T
- la **longueur** du tour est 4
- T est appelé un **(v_1, v_2)-tour**

La séquence $C = v_1, e_1, v_2, e_3, v_6$ est un **chemin** de G; c'est un **chemin-(v_1, v_6).** Mais la séquence T ci-dessus n'est pas un chemin de G.

v_1 et v_6 sont **connectés** car il existe un **(v_1, v_6)-chemin**

G n'est pas connecté car tous ses sommets ne sont pas connectés deux à deux, par exemple v_5 n'est connecté à aucun autre sommet de G.

Soit $W = \{v_2, v_3, v_4, v_6\}$ G-W est le graphe de la figure III.3.

v1

v5

Fig. III.3.

Le **sous-graphe induit** par W, **G(W)** est repris à la figure III.4.

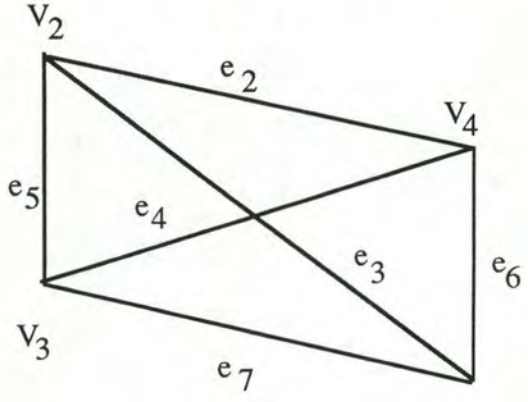


Fig. III.4

et G(W) forme une clique.

B. Énoncé du problème de clique proprement dit

Quand nous parlons du problème de clique, nous faisons référence à un **problème d'optimisation** combinatoire bien connu qui est le suivant :

> "Étant donné un graphe non orienté défini par un ensemble de sommets V et un ensemble d'arêtes E, déterminer une clique ayant le nombre maximum de sommets. Autrement dit, étant donné G = (V, E), trouver une clique maximum dans G, c'est-à-dire trouver le plus grand sous-graphe complet contenu dans G." (Melhorn K., 1984)

A ce problème d'optimisation peut être associé un **problème de décision** qui s'énonce comme suit :

> "Etant donné un graphe non orienté G = (V, E), à n sommets, et un entier k, existe-t-il une clique de taille k dans G, c'est-à-dire peut-on trouver $V' \subseteq V$, avec $|V'| = k$, tel que $(u, v) \in E$ pour tout $u, v \in V'$" (Melhorn K., 1984).

En fait, il existe un algorithme trivial, mais non efficace (la notion d'efficacité sera détaillée dans le point IV.B) pour solutionner ce problème de clique. Ce problème stipule qu'il suffit de générer tous les sous-ensembles de sommets $V' \subseteq V$ de cardinalité k et de regarder si le sous-graphe induit G(V') forme une clique.

Il y a $\binom{n}{k}$ sous-ensembles V' avec k éléments ($n = |V|$). Par conséquent, nombreux sont les candidats pour une clique de taille k. Vérifier si un sous-ensemble de sommets $V' \subseteq V$ forme une clique, c'est-à-dire vérifier si un candidat est vraiment une solution, se fait alors assez facilement. Toutefois, la seule façon connue pour trouver une solution au problème de décision est de faire une recherche exhaustive (voir point IV.D.2) parmi tous les candidats. Pour un entier $k = n/2$, l'algorithme trivial doit contrôler $\binom{n}{n/2} > 2^n/(n + 1)$ sous-ensembles. On peut conclure que cet algorithme possède un temps d'exécution au moins égal à $2^n/(n+1)$, donc un temps qui croît exponentiellement avec la taille des données (Melhorn K., 1984).

Cela montre que le problème de clique appartient à la classe des problèmes NP-complets (non déterministe polynomial). Il n'existe pas d'algorithme asymptotiquement efficace pour résoudre ce genre de problème. Nous expliquerons cela au point IV.B.

C. Quelques applications du problème de clique

Le problème de clique possède de nombreuses applications pratiques. En effet, le problème de clique équivaut à chercher dans un ensemble d'éléments quelconques, le plus grand sous-ensemble d'éléments, partageant tous, deux à deux, une même relation.

1. Exécution de projets

Les éléments peuvent, par exemple, être des projets qui doivent être exécutés. Cet exemple est tiré du livre "Graph Theory : An Algorithmic Approach" p. 32. (Christofides N., 1975).

Notons que l'exemple cité concerne l'existence d'un ensemble de sommets indépendant maximal ("independent set"). Cependant, ce problème est aisément transposable en termes de clique maximale. En effet, chercher un ensemble de sommets indépendant maximal dans un graphe G revient à chercher une clique dans le graphe complémentaire. $\overline{G}$.

Nous vous proposons ici la version transposée de l'exemple précité :

Considérons n projets qui doivent être exécutés et supposons que chaque projet x_i nécessite un sous-ensemble $R_i \subseteq \{1, \ldots p\}$ de p ressources disponibles pour son exécution. Supposons aussi que chaque projet (étant donné ses besoins en ressource) est exécuté en une seule période de temps. Nous pouvons former un graphe G où chaque sommet correspond à un projet et où une arête (x_i, x_j) est ajoutée chaque fois que x_i et x_j n'ont pas de ressource en commun, c'est-à-dire chaque fois que $R_i \cap R_j = 0$. Une clique de G représente dès lors, un ensemble maximal de projets qui peuvent être exécutés simultanément pendant une seule période de temps.

2. Analyse conformationnelle des protéines

Nous avons vu que le concept de clique était également applicable en biologie.

J. C. Hempel a montré que le concept abstrait de clique correspondait au concept de famille conformationnelle. Une famille conformationnelle est en fait un ensemble de conformères (structures tridimensionnelles de protéines) où chaque membre est similaire à tous les autres. La similarité est évaluée par la mesure de la déviation RMS ("root mean square") des coordonnées cartésiennes d'un ensemble d'atomes spécifiés dans la définition de la famille. Les sommets du graphe sont les conformères, et une arête est présente entre deux sommets si la déviation RMS entre les coordonnées cartésiennes est inférieure à un seuil préétabli. La formule du RMS est :

$$RMS = \sqrt{\sum_{i=1}^{n} \frac{(x1-x2)^2+(y1-y2)^2+(z1-z2)^2}{n}}$$

3. Comparaison de structures tridimensionnelles

H. M. Grindley (Grindley H.M., Artymiuk P.Y., Rice D.W., Willet P., 1993) et ses collaborateurs utilisent l'algorithme de clique de C. Bron et J. Kerbosch (voir point IV.F.1) dans un programme qui permet de comparer des paires de structures tridimensionnelles de protéines pour identifier les motifs d'éléments de structures secondaires qu'elles ont en commun. Les éléments de structures secondaires constituent les sommets du graphe et les relations spatiales et angulaires existant entre eux constituent alors les arêtes du graphe.

De manière plus détaillée, l'approche par la détection de cliques pour identifier les sous-graphes maximaux tridimensionnels, implique l'identification de cliques dans un "graphe de correspondance" c'est-à-dire une structure de données qui contient toutes les équivalences possibles entre les deux graphes qui sont comparés. Ces deux graphes représentent les structures secondaires des deux protéines comparées. Ce sont des graphes dont les arêtes sont étiquetées; c'est-à-dire qu'on leur a attribué une valeur qui correspond aux relations spatiales et angulaires existant entre les structures secondaires.

Étant donnée une paire de graphes A et B, un graphe de correspondance C, peut être formé par le processus suivant :

Etape 1 : Créer l'ensemble de toutes les paires de sommets (a_i, b_j), où a_i est un sommet provenant du graphe A, et bj un sommet provenant du graphe B, de telle façon que les sommets de chaque paire soient du même type (hélice α ou brin β).

Etape 2 : Former le graphe C, dont les sommets sont les paires de l'étape 1. Les sommets du graphe de correspondance (a_{i_k}, b_{j_l}) et (a_{i_m}, b_{j_n}) sont joints par une arête dans C si la valeur de l'arête entre a_{i_k} et a_{i_m} est la même que celle de l'arête entre b_{j_l} et b_{j_n} plus ou moins un certain seuil.

Etape 3 : Les sous-graphes communs maximaux correspondent donc aux cliques du graphe de correspondance.

4. Fixation de petites molécules à un site récepteur

L'algorithme de C. Bron et J. Kerbosch est aussi utilisé par F. S. Kuhl (Kulh F.S., Crippen G.M., Friesen DK., 1984) et ses collaborateurs pour résoudre le problème de prédiction du mode de fixation d'une petite molécule à un site récepteur. Pour ce problème, ils montrent que deux approches sont possibles. Une première approche est, étant donné une petite molécule rigide et sa géométrie, de rechercher directement son orientation dans l'espace qui maximise le degré de contact. Une deuxième approche est une approche combinatoire dans laquelle seul le critère "contact-no-contact" est considéré. Cette approche qui considère un modèle "tout-ou-rien" permet d'utiliser un algorithme de clique. Dans cet exemple, les sommets du graphe sont des paires ordonnées où le premier élément est l'un des points de fixation du site récepteur et le second un point de la molécule. Une arête est présente entre deux sommets (s_i, m_i) et (s_k, m_l) si la distance entre les points du site, $D(s_i, s_k)$ est égale à la distance entre les points de la molécule $D(m_i, m_l)$ plus ou moins un certain seuil. Donc étant donné (S, M) un

problème de fixation ("docking problem") où S est un site de fixation et M une molécule qui se fixe et étant donné le graphe de fixation ("docking graph") correspondant, chaque ensemble $\{(s_{i1}, m_{i1});\ldots (s_{ik}, m_{ik})\}$ de (S, M) qui correspond à un ensemble maximal de contacts entre des points du site et des points de la molécule (appelé un "maximal matching") est une clique de G.

D. Définition de notre problème sous forme de graphe

Notre problème peut être décrit sous forme de graphe. Nous pouvons considérer en effet, que les sommets du graphe sont toutes les fenêtres mobiles similaires à une fenêtre initiale. Elles correspondent, rappelons-le, à tous les "matchs" obtenus pour une fenêtre initiale, prise dans une des séquences à aligner. Ils sont trouvés dans des séquences différentes de celle-là (voir point II.B). Les arêtes du graphe indiquent que les segments correspondant à leurs extrémités sont similaires. La mesure de similarité est expliquée dans la description de la méthode d'alignement.

D'après cette définition du graphe et la définition d'un "match" complet donnée au point II.B.3, une clique est toujours associée à un "match" complet.

Trouver tous les "matchs" complets revient à rechercher toutes les cliques d'une cardinalité donnée dans un graphe.

Remarquons qu'une clique ne peut contenir plusieurs segments d'une même séquence. En effet, puisque pour appartenir à un "match" complet, les segments, en plus de former un lien complet, doivent appartenir à des séquences différentes. Autrement dit, une arête ne sera jamais présente entre deux sommets représentant des segments d'une seule séquence. Par conséquent, nous ne pourrons jamais trouver deux sommets issus d'une même séquence dans une clique (maximum ou non) et donc, nous n'aurons jamais de clique de taille supérieure au nombre de séquences à aligner avec la fenêtre initiale.

D'autre part, comme notre but est d'aligner toutes les séquences, il faut donc trouver un segment dans chaque séquence de protéines.

Nous nous intéresserons dès lors, uniquement à la recherche des cliques de taille maximum, de cardinalité égale au nombre de séquences à aligner avec la fenêtre initiale (= le nombre de séquences prises pour réaliser l'alignement moins 1).

Nous pouvons formuler notre problème de manière précise, comme suit :

" Étant donné un graphe G = (V, E), où l'ensemble des sommets représente des segments de séquences de protéines, tous similaires à un même segment d'une séquence de protéine et, où E est l'ensemble des arêtes (une arête est présente entre deux sommets s'ils sont similaires et qu'ils représentent des segments appartenant à des séquences différentes); rechercher toutes les cliques d'une taille maximum donnée, égale au nombre de séquences à aligner - 1. Ces cliques correspondent aux "matchs" complets dans la méthode d'alignement."

D'après la définition du graphe donnée ci-dessus, nous pouvons également dire que pour réaliser un alignement complet, nous devrons analyser un tel graphe pour chaque fenêtre initiale.

Illustrons cette définition du problème par un exemple :

Soient I, A, B, C, quatre séquences que l'on essaie d'aligner.

Dans une étape du "matching", une fenêtre initiale I_0 dans la séquence I est comparée aux fenêtres mobiles baladées le long des autres séquences au cours de la procédure de "scanning". Les segments similaires à cette fenêtre initiale trouvés dans la séquence A sont les "matchs" A_1, A_2, A_3 et A_4, ceux trouvés dans la séquence B, sont B_1, B_2, B_3, B_4, B_5, et ceux trouvés dans C sont C_1, C_2, C_3, C_4, C_5 et C_6.

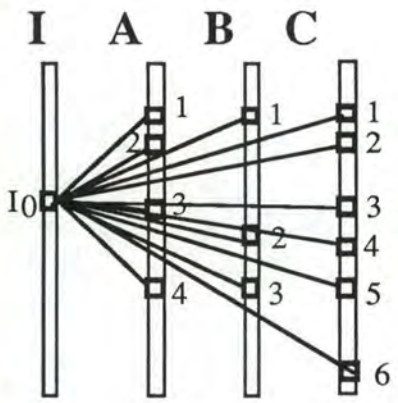


Fig. III.5.

Dans la méthode de la distance minimum, la détection d'un "match" complet, se fait en recherchant, parmi tous les "matchs" d'une séquence, celui qui est le plus similaire à I_0.

Supposons que le graphe modélisant cette situation est le suivant :

$G = (V, E)$

avec $V = \{A_1, A_2, A_3, A_4, B_1, B_2, B_3, B_4, B_5, C_1, C_2, C_3, C_4, C_5, C_6\}$

et $E = \{(A_1, B_2), (A_1, B_3), (A_1, B_4), (A_1, C_3), (A_3, B_3), (A_3, B_1), (B_1, C_1), (B_1, C_4), (B_2, C_6), (B_5, C_5), (B_5, C_6)\}$

1. La méthode de la distance-minimum,

- recherche d'abord le "match" le plus similaire à I_0 dans chaque séquence;
  - dans A, c'est A_1;
  - dans B c'est B_2;
  - dans C c'est C_3;
- ensuite, elle teste si A_1, B_2, C_6 forment un lien complet, c'est-à-dire si A_1 est similaire à B_2, donc s'il existe une arête (A_1,B_2)et C_3, une arête (A_1,C_3) et si B_2 est similaire à C_3 (B_2,C_3).

Dans ce graphe, A_1 est similaire à B_2 et C_3, mais B_2 n'est pas similaire à C_3. La méthode de la distance-minimum ne trouvera pas de "match" complet pour la fenêtre initiale I_0.

2. La méthode de recherche des cliques devra retrouver tous les "matchs" complets; c'est-à-dire qu'elle devra regarder si tous les sous-ensembles constitués exactement de trois sommets, issus de chaque séquence, forment une clique. En d'autres termes, elle devra extraire toutes les cliques d'une taille maximum égale à 3. Les cliques trouvées seront, dans ce cas-ci :

— A1, B2, C6;
— A3, B1, C1;

Cette stratégie permet de trouver 2 "matchs" complets.

IV. Résolution du problème des cliques en informatique

A. Représentation d'un graphe

Un **graphe** est un objet mathématique qui modélise précisément certaines situations.

Pour traiter des graphes à l'aide d'un programme d'ordinateur, il faut tout d'abord décider comment les représenter dans le programme.

Il existe deux représentations communément utilisées. Le choix d'une représentation dépend, premièrement de la densité en arêtes du graphe et deuxièmement, de la nature des opérations à réaliser.

La représentation la plus directe pour les graphes est celle appelée **matrice d'adjacence**. C'est un tableau à deux dimensions. Disons mat(n, n), (où n = |V| et V est l'ensemble des sommets du graphe). Ce tableau est rempli de valeurs booléennes telles que mat(i, j) = 1, si i et j sont des sommets adjacents (c'est-à-dire joints par une arête) et 0 sinon. Remarquons que chaque sommet est en réalité représenté par deux bits : une arête reliant i et j est représentée par la vraie valeur à la fois au niveau de l'élément mat(i, j) et mat(j, i). De l'espace peut donc être épargné, en stockant seulement la moitié de cette matrice symétrique. Toutefois, cela n'est pas toujours pratique et les algorithmes sont plus simples avec la matrice entière. En pratique, on considère habituellement qu'il existe une arête partant de chaque sommet vers lui-même. Donc mat(i, i) est mis à 1 pour tout $i \in V$. Dans certains cas il est plus approprié de mettre la diagonale à 0. Pour des graphes avec des poids attribués aux arêtes, il suffit de remplacer la place des valeurs booléennes de la matrice d'adjacence par des poids.

Une autre mise en oeuvre, plus adaptée pour les graphes peu denses, est la **liste d'adjacence**. A chaque sommet du graphe, est associée une liste de tous les autres sommets qui lui sont adjacents. L'ordre des sommets dans la liste d'adjacence affecte l'ordre dans lequel les sommets sont traités par l'algorithme.

B. Complexité des algorithmes

Lors de l'estimation de l'efficacité des algorithmes, on se base principalement sur la quantité de ressources (temps et espace) qu'ils requièrent en fonction de la "taille" naturelle (quantité des données traitées) des problèmes solutionnés. Cette taille est souvent appelée **n**, ou **N**.

Généralement, on s'intéresse à la durée qu'un programme pourrait prendre, en moyenne, pour traiter des entrées "typiques". On s'intéresse aussi au temps qu'il mettrait dans le cas le

plus défavorable où la plus mauvaise configuration d'entrée est mise en oeuvre (Sedgewick R., 1990).

Virtuellement, tous les algorithmes possèdent un temps d'exécution maximal proportionnel à une des fonctions suivantes (Sedgewick R., 1990) :

1, le temps d'exécution est dit **constant**. La plupart des instructions du programme sont exécutées une seule fois, ou seulement un très petit nombre de fois.

$\log N$, le temps d'exécution est dit **logarithmique** quand le programme devient légèrement plus lent lorsque N augmente. Ce temps d'exécution est souvent caractéristique du programme qui solutionne un grand problème en le transformant en un problème plus petit.

N, le temps d'exécution est dit **linéaire** s'il augmente linéairement avec N. C'est généralement le cas lorsqu'une petite quantité de calcul est réalisée sur chaque élément d'entrée.

$N\log N$, le temps d'exécution est dit "***N*log*N***", ce qui est caractéristique d'un algorithme qui solutionne un problème en le divisant en sous-problèmes plus petits, solutionnant ceux-ci indépendamment et combinant les solutions.

N^2, le temps d'exécution est dit **quadratique**. Ces algorithmes sont alors utilisables, en pratique, seulement pour des problèmes de taille relativement petite.

N^3, le temps d'exécution est dit **cubique**, ces algorithmes ne sont utilisables, en pratique, que pour des problèmes de petites tailles.

2^N, le temps d'exécution est dit **exponentiel**. Ces algorithmes ne sont utilisables, en pratique, que pour des problèmes de taille très petite. Lorsque la taille du problème double, le temps d'exécution est, lui, élevé au carré.

Le temps d'exécution d'un programme sera probablement une constante multipliée par un de ces termes (les "termes dominants") plus des termes plus petits. Les valeurs des coefficients constants et des termes inclus dépendent des résultats des analyses et des détails d'implantation. Pour de grandes valeurs de N, l'effet des termes dominants est prépondérant. Pour des petites valeurs de N ou pour des algorithmes construits avec précautions, un plus grand nombres de termes peuvent contribuer à l'évaluation du programme, et les comparaisons des algorithmes sont alors plus difficiles. Dans la plupart des cas, les temps d'exécution des programmes sont caractérisés de "linéaires", "cubiques", ... Quelques autres fonctions se rencontrent aussi. Par exemple, un algorithme qui a N^2 entrées et un temps d'exécution cubique en N, est mieux classé comme étant un $N^{3/2}$ algorithme (Sedgewick R., 1990).

Afin d'analyser la **performance d'algorithmes**, une étude de leur comportement dans les cas les plus défavorables peut être utilisée, ignorant les facteurs constants, pour déterminer la dépendance fonctionnelle du temps d'exécution (ou d'une autre mesure) sur le nombre d'entrées (ou d'une autre variable). Cette approche permet de faire abstraction des caractéristiques d'implantation. L'artéfact mathématique pour exprimer précisément la notion "est proportionnel à" est appelée la **O-notation**, ou **"grand-oh notation"** (**"big-oh notation"**) définie comme suit (Sedgewick R., 1990) :

> Notation : Une **fonction t(N)** est dite **O(f(N)),** s'il existe des constantes c_0 et N_0 telles que t(N) est inférieur à $c_0 f(N)$ pour tout $N > N_0$.

Informellement, cette notation englobe la notion "est proportionnel à" et permet à l'analyste de ne pas se soucier des détails des caractéristiques d'une machine particulière. De plus cette notation est indépendante des entrées. Donc la O-notation permet d'établir une limite supérieure pour le temps d'exécution indépendante des entrées et des détails d'implantation.

Étant donné un codage et un algorithme, la **fonction de complexité temporelle**, .ou la **fonction du temps d'exécution**. d'un algorithme f : N → N, exprime le temps maximum f(n) nécessaire pour solutionner n'importe quelle occurrence d'un problème dont la longueur d'encodage vaut au plus N (Grötschel M., Lovasz L., Schrijver A., 1988).

Pour une machine de Turing, le "temps" signifie le nombre d'étapes nécessaires pour que la machine atteigne l'état final E à partir de l'état initial B, pour une certaine chaîne de caractères en entrée (Réf 56).

De façon similaire, la **fonction de complexité spatiale**.g : N → N. d'un algorithme exprime l'espace maximum nécessaire g(n) pour solutionner n'importe quelle occurrence d'un problème dont la longueur d'encodage vaut au plus N (Réf 56).

Dans la "t-tape Turing Machine Model", l'espace signifie la longueur maximale des chaînes de caractères apparaissant tout au long des exécutions des étapes sur la "tape" i, sommée sur i = 1 jusqu'à t (Grötschel M., Lovasz L., Schrijver A., 1988).

C. Algorithmes polynomiaux et non déterministes polynomiaux

Un **algorithme polynomial**, c'est-à-dire qui possède un temps polynomial est un algorithme dont la fonction de complexité temporelle f(n) satisfait l'inégalité $f(n) \leq p(n)$ pour tout $n \in N$, pour un polynôme p (Grötschel M., Lovasz L., Schrijver A., 1988).

La **classe P** comprend l'ensemble des problèmes pouvant être résolus par un algorithme déterministe qui a un temps d'exécution polynomial (Sedgewick R., 1990).

Qu'entend-on par **algorithme déterministe** ? Appelons **état** d'un algorithme la combinaison de l'emplacement de l'instruction actuellement exécutée avec les valeurs actuelles de toutes les variables. Un algorithme est déterministe si, quel que soit l'état qu'il a atteint, il y a

au plus un état qu'il puisse atteindre consécutivement (Golombic, 1980). Un algorithme déterministe ne peut donc accomplir qu'une seule chose à la fois.

La **classe NP** comprend l'ensemble des problèmes qui peuvent être solutionnés par un algorithme non déterministe qui possède un temps d'exécution polynomial (Garey M. R. et Johnson, 1979) (Sedgewick R., 1990).

Qu'entend-on par algorithme **non déterministe**?

Un algorithme est non déterministe si, ayant atteint un certain état, il peut atteindre plus d'un état et poursuivre simultanément sur chacun des états suivants (Golombic, 1980).

Par conséquent, alors qu'un algorithme déterministe doit explorer une seule alternative à la fois parmi un ensemble d'alternatives, un algorithme non déterministe examine toutes les alternatives au même moment (Golombic, 1980).

Les algorithmes non déterministes ne sont en aucun cas de nature probabiliste, ce sont des algorithmes qui peuvent être dans plusieurs états simultanément (Garey M. R. et Johnson D. S., 1979). La classe P est incluse dans la classe NP ($P \subseteq NP$).

Pour les problèmes de la classe NP, aucun algorithme asymptotiquement efficace; c'est-à-dire utilisable pour de grandes tailles de données, n'a jamais encore été trouvé.

Un algorithme asymptotiquement efficace est polynomial pour des petites valeurs de k. La classe des algorithmes polynomiaux comprend tous les algorithmes asymptotiquement efficaces. Donc, si un problème ne peut être résolu par un algorithme polynomial, à plus juste raison, il ne pourra être résolu par aucun algorithme asymptotiquement efficace.

Les problèmes NP sont parfois aussi qualifiés de problèmes **NP-durs** et de problèmes **NP-complets**.

Un problème Π est dit NP-dur si un algorithme déterministe polynomial qui le solutionne peut être utilisé pour obtenir un algorithme polynomial déterministe pour chaque problème appartenant à la classe NP (Garey M. R. et Johnson D. S., 1979). En d'autres mots, Π est NP-dur si il est au moins aussi dur que n'importe quel problème dur de la classe NP (Garey M. R. et Johnson D. S., 1979).

Un problème NP-durs dans NP est appelé NP-complets. Ce type de problème est au moins aussi *dur* que n'importe quel problème de la classe NP, souvent plus dur que les problèmes de cette classe (Garey M. R. et Johnson D. S., 1979).

Une autre définition d'un problème NP-complets est la suivante : un problème Π est NP-complets s'il appartient à NP et si tous les autres problèmes dans NP peuvent être transformés en un temps polynomial en Π (Grötschel M., Lovasz L., Schrijver A., 1988). Autrement dit, si tous les problèmes de la classe NP lui sont polynomialement réductibles.

Donc, chaque problème Π, NP-complet, possède la propriété suivante : si Π peut être solutionné en un temps polynomial, alors tous les problèmes de NP peuvent l'être également (Grötschel M., Lovasz L., Schrijver A., 1988). C'est-à-dire, si Π est NP-complet et si $\Pi \in P$,

alors nous avons l'égalité suivante, P = NP (Grötschel M., Lovasz L., Schrijver A., 1988). Cela signifierait que tous les problèmes solutionnés par un algorithme exponentiel pourraient être sélectionnés par un algorithme déterministe polynomial (Garey M. R. et Johnson D. S., 1979). Cette dernière proposition n'étant pas plausible, nous en déduisons que les algorithmes déterministes polynomiaux ne donnent pas de solutions aux problèmes NP durs ou NP-complets (Garey M. R. et Johnson D. S., 1979).

Pour la classe des problèmes NP-complets, aucun algorithme asymptotiquement efficace n'a été trouvé jusqu'à présent. Dans ce domaine, on ne travaille qu'avec des algorithmes du type exponentiels.

Le problème de clique est un problème NP, et plus précisément NP-complet. La preuve de ce théorème est donnée dans "Graph Algorithms and NP-completeness", Kurt Mehlhorn (1984), p. 198.

D. Résolution des problèmes NP-complets

Les problèmes NP-complets sont probablement très difficiles à solutionner.

Néanmoins, étant donné que nous les rencontrons fréquemment dans la pratique, et donc qu'il est nécessaire de les solutionner, on se base sur un ensemble de différentes méthodes pour les résoudre.

1. Approches de résolution

Cas spéciaux : on réexamine le problème de façon visuelle, puis l'on vérifie si le problème NP-complets est réellement à solutionner dans toute sa généralité, ou si l'on peut se contenter de solutionner un cas spécial. L'avantage est que le cas spécial peut avoir une solution en un temps polynomial (Melhorn K., 1984, pp 208 et 209).

Programmation dynamique et technique "Branch-and-Bound" : la programmation dynamique et le "Branch-and-Bound" sont deux techniques qui peuvent être appliquées à la plupart des problèmes NP-complets. Ces deux techniques sont essentiellement des variantes intelligentes de recherche exhaustive (Melhorn K., 1984, pp 208 et 209).

Analyse de probabilités : des analyses de probabilités peuvent parfois montrer que les occurrences "difficiles" d'un problème NP-complets sont assez rares. Il est alors possible de concevoir des algorithmes avec des temps d'exécution appréciables. Bien sûr, il subsiste toujours le problème de la justification de la distribution de probabilité postulée pour les occurrences du problème (Melhorn K., 1984, pp 208 et 209).

Algorithme d'approximation : des algorithmes d'approximation peuvent parfois aboutir à de très bonnes solutions en un temps court (Melhorn K., 1984, pp 208 et 209).

Heuristiques : des algorithmes heuristiques peuvent parfois, pour une raison inconnue, donner des résultats satisfaisants. (Melhorn K., 1984, pp 208 et 209).

2. Recherche exhaustive dans un graphe

La recherche exhaustive est une méthode de résolution fréquemment utilisée dans le problème de clique. C'est pourquoi, nous allons introduire cette méthode de résolution sur base du chapitre 44, "Exhaustive Search", du livre "Algoritms in C" de Robert Sedgewick ainsi que sur le livre "Structure de données et algorithmes" d'Alfred Aho, John Hopcroft et de Jeffrey Ullman aux pages 332-340.

Pour certains problèmes, il est nécessaire de rechercher une solution exacte parmi un grand nombre de solutions potentielles. Dans une recherche exhaustive, toutes les solutions potentielles sont examinées afin de vérifier si elles correspondent, ou non, à la solution exacte du problème (processus de décision ou "decision-making process").

Il n'existe pas, semble-t-il, d'algorithme efficace pour solutionner les problèmes qui nécessitent une recherche exhaustive.

Toutefois, il est parfois possible de réduire considérablement le nombre de possibilités contrôlées, en essayant de découvrir les décisions incorrectes le plus rapidement possible dans le processus de décision.

a. Technique dite du "Backtracking"

La méthode de **recherche avec rebroussement** ("**backtracking**"), .est une technique systématique de recherche exhaustive, c'est-à-dire donnant toutes les solutions. Elle résout un problème en générant systématiquement toutes les solutions possibles. Le processus de "backtracking" peut être décrit par un arbre de recherche exhaustive dont les noeuds correspondent à des solutions partielles qui peuvent être successivement augmentées, de manière récursive, pour produire une solution complète. Descendre dans l'arbre correspond à progresser vers une solution plus complète. Remonter dans l'arbre correspond à *faire marche arrière* ("backtrack") vers une solution partielle générée précédemment, à partir de laquelle il serait probablement intéressant de revenir pour une nouvelle progression. Autrement dit, la technique de "backtracking" explore une branche de l'arbre et, lorsqu'elle arrive dans un *cul de sac*, repart du dernier point de bifurcation.

Le temps pris par une procédure de recherche exhaustive pour explorer l'arbre de toutes les possibilités ("exhaustive search tree") est proportionnel au nombre de noeuds qu'il contient. Cette durée sera importante dans le cas de traitement de grands arbres.

Grâce à certaines techniques, il est possible de réduire, de façon relativement importante, le nombre de possibilités essayées, c'est-à-dire le nombre de noeuds qui doivent être explorés

dans l'arbre. Ces techniques *élaguent* ("prune") l'arbre de toutes les possibilités, en coupant certaines branches, par la suppression de certains noeuds de l'arbre.

L'algorithme étant récursif, le fait de supprimer un noeud annule la recherche dans tous le sous-arbre de ce noeud-racine. Pour des arbres importants, c'est un gain non négligeable de temps. Il faut donc le faire autant que possible pour éviter de visiter les sous-arbres inutiles. Une coupure haute dans l'arbre peut conduire à des économies significatives de temps de traitements.

Une technique importante d'élagage consiste en la suppression de symétries : il s'agit d'empêcher qu'une même solution soit trouvée plusieurs fois.

b. Technique "branch-and-bound"

La technique de **"branch-and-bound"** est une autre technique d'élagage ("pruning") d'un arbre de recherche exhaustive. Dans cette technique, chaque direction de recherche est déterminée en fonction de contraintes spécifiques. Ainsi, en tout noeud *n* de l'arbre auquel on a abouti, on essaie d'*aiguiller* ("branch") la suite de la recherche vers une ramification particulière de l'arbre, après avoir calculé un coût minorant, donc une borne locale ("bound"), et l'avoir comparé à la borne globale obtenue jusqu'à ce stade pour le problème. Ainsi, cette méthode calcule des limites ("bounds") sur les solutions partielles dans le but de limiter le nombre de solutions entières devant être examinées.

La méthode "branch-and-bound" peut être combinée à un algorithme de "backtracking". Toutefois, malgré un critère sophistiqué, il est généralement vrai que le temps d'exécution d'un algorithme de "backtracking" reste exponentiel.

Succinctement, si chaque noeud dans l'arbre de recherche a α enfants, en moyenne, et que la longueur du chemin-solution est N, alors, le nombre attendu de noeuds dans l'arbre sera proportionnel à a^N.

Des **règles heuristiques** sont parfois utilisées comme conditions pour stopper la recherche dans une branche de l'arbre.

Les différentes règles de "backtracking" ont pour objectif de réduire la valeur de α, c'est-à-dire réduire le nombre de choix à faire à chaque noeud. Cet élément à tout son intérêt, car une réduction du nombre d'enfants à rechercher, permet de solutionner des problèmes de plus grande taille.

Exemple : Un algorithme avec un temps d'exécution proportionnel à 1.1^N peut solutionner un problème peut-être 8 fois aussi grand qu'un algorithme qui a un temps d'exécution proportionnel à 2^N. D'un autre côté, comme mentionné supra, aucun de ces algorithmes ne marche valablement sur de très grands problèmes.

3. Recherche exhaustive et problème de clique

Expliquons la **recherche avec rebroussement** ("backtracking") dans le cas de la recherche de cliques (Garey M.R. & Johnson D.S., 1979).

Soit le graphe G = (V, E)

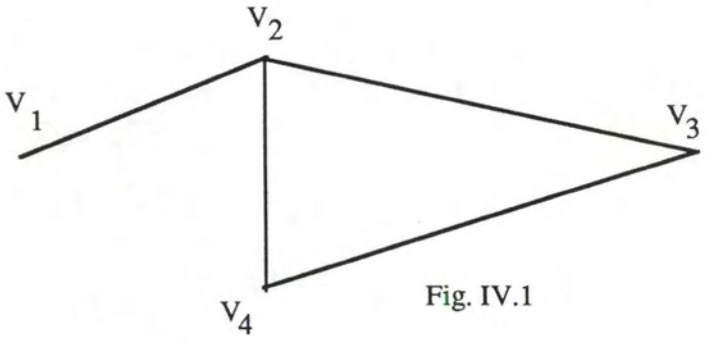


Fig. IV.1

Une recherche avec rebroussement sans contrainte (il n'y a aucun élagage réalisé) produit l'arbre de toutes les possibilités suivantes :

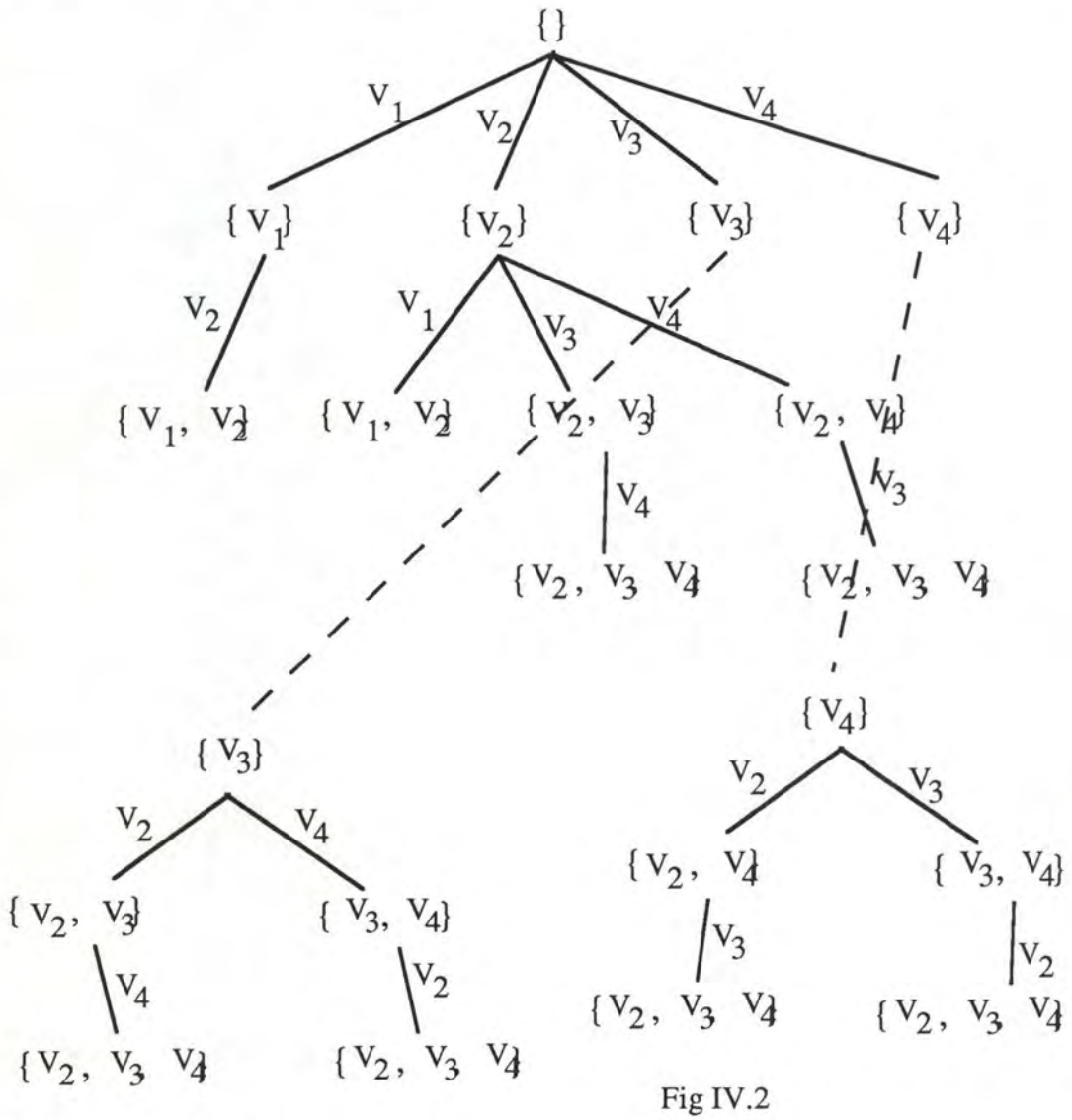


Fig IV.2

Chaque noeud dans l'arbre de recherche correspond à un sous-graphe complet, et chaque arête correspond au choix d'un sommet du graphe.

Un fils d'un noeud donné C est obtenu en ajoutant à C un sommet $x \notin C$ qui est adjacent à chaque sommet formant C. L'arête de C au fils $C \cup \{x\}$ correspond au sommet x.

Dans cette recherche sans contrainte, chaque clique est générée plusieurs fois. $\{v_1,v_2\}$ et $\{v_2,v_3,v_4\}$ sont générées 2 fois et 6 fois respectivement.

En général, une clique de taille k est générée (k!) fois. Toutes les arêtes de l'arbre de la fig. IV.2 peuvent être élaguées en utilisant les théorèmes suivants :

Théorème 1

Soit S, un noeud de l'arbre A (c'est-à-dire un sous-ensemble de sommets de G qui forme un sous-graphe complet de G), et soit $S \cup \{x\}$, le premier fils de S dans A qui est exploré (x doit être adjacent à chaque sommet compris dans S). Supposons que tous les sous-arbres du noeud $S \cup \{x\}$ dans A aient été explorés, donc que toutes les cliques contenant $S \cup \{x\}$ aient été générées. Alors, parmi les fils $S \cup \{v\}$ de S, seulement ceux pour lesquels $v \notin Adj(x)$ doivent être explorés. (voir fig. IV.3).

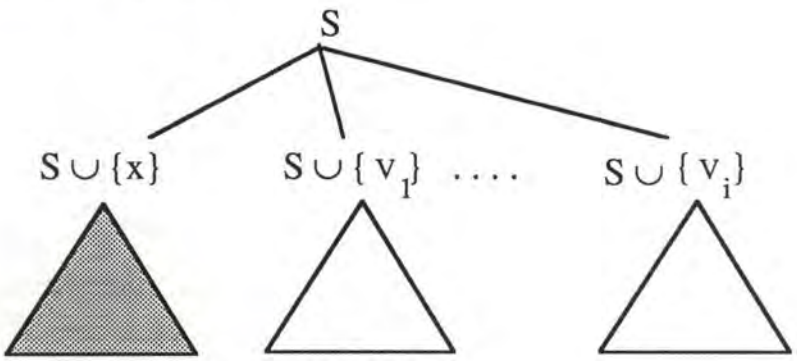


Fig IV.3

Fig. IV.3. Selon le théorème 1, le sous-arbre dont la racine est $S \cup \{v_i\}$ ne doit pas être exploré si le sous-arbre dont la racine $S \cup \{x\}$ (en grisé) a été exploré et que $v_i \in Adj(x)$.

Preuve :

Soit C, une clique générée en explorant un sous-arbre dont la racine est $S \cup \{v_i\}$, où $v_i \in Adj(x)$. Évidemment, $S \subseteq C$, et si C contient un sommet $v_j \in Adj(x)$, alors C sera trouvé quand le sous-arbre, dont la racine est $S \cup \{v_j\}$ est exploré (remarquons que $S \subseteq Adj(x)$, donc $v_j \notin S$). Si C ne contient pas un tel sommet v_j, alors il doit être trouvé quand le sous-arbre dont la racine est $S \cup \{x\}$ a été exploré.

Remarquons que le Théorème 1 ne peut être appliqué itérativement à l'arbre lorsque les fils d'un noeud S sont examinés, c'est-à-dire que, après l'utilisation du théorème 1 pour élaguer un sous-arbre dont la racine est $S \cup \{v_i\}$ (voir fig. IV.2), comme $v_i \in Adj(x)$, on ne peut élaguer le sous-arbre dont la racine est $S \cup \{v_k\}$ quand $v_k \in Adj(v_j)$ pour un certain $v_j \in Adj(x)$.

En d'autres mots, le théorème 1 ne s'applique qu'au premier fils de S qui est exploré et pas à ses autres fils. Toutefois, le théorème trivial 2 donne une autre solution.

Théorème 2

Soit S un noeud dans l'arbre de recherche A et soit $S' \subset S$, S' étant un ancêtre propre de S dans A. Si tous les sous-arbre du noeud $S' \cup \{x\}$ ont été explorés, de telle façon que toutes les cliques contenant $S' \cup \{x\}$ aient été générées alors, les sous-arbres non explorés ayant $S \cup \{x\}$ comme racine, peuvent être ignorés.

C. Bron et J. Kerbosch proposent une procédure de "backtracking" se basant sur ces 2 théorèmes pour générer toutes les cliques d'un graphe en élaguant l'arbre de recherche (voir point IV.F). Ces deux théorèmes permettent de ne générer qu'une seule fois les cliques.

E. Définition de notre problème sous forme informatique

Le problème informatique associé à notre problème biologique est de trouver un algorithme "efficace" pour la recherche de toutes les cliques d'une taille maximum donnée, dont la valeur est égale au nombre de séquences alignées moins 1, dans le graphe formé par les segments qui "match" avec une fenêtre initiale (voir point III.D).

Comme nous l'avons vu précédemment, deux représentations principales d'un graphe permettent de traiter celui-ci au moyen d'un algorithme. Nous utiliserons la représentation du graphe par la matrice d'adjacence.

La définition d'un "match" complet telle qu'elle a été donnée dans la présentation de la méthode d'alignement de séquences, implique les considérations suivantes. Soit un ensemble de s segments ayant chacun le même nombre d'acides aminés w. Nous pouvons associer, à cet ensemble, une matrice r symétrique (s x s) d'adjacence, avec $r_{ij} = 1$ si le segment i "matche" avec le segment j (i et j appartenant à des séquences différentes), et $r_{ij} = 0$ sinon.

Voici la représentation, sous forme de matrice d'adjacence, de l'exemple donné dans le point III.D :

où $G = (V, E)$

avec $V = \{A_1, A_2, A_3, A_4, B_1, B_2, B_3, B_4, B_5, C_1, C_2, C_3, C_4, C_5, C_6\}$

et $E = \{(A_1, B_2), (A_1, B_3), (A_1, B_4), (A_1, C_3), (A_3, B_3), (A_3, B_1), (B_1, C_1), (B_1, C_4), (B_2, C_6), (B_5, C_5), (B_5, C_6)\}$

	A1	A2	A3	A4	B1	B2	B3	B4	B5	C1	C2	C3	C4	C5	C6
A1	1	0	0	0	0	1	1	1	0	0	0	1	0	0	0
A2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
A3	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0
A4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
B1	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0
B2	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
B3	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
B4	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
B5	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1
C1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
C2	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
C3	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
C4	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
C5	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0
C6	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1

L'algorithme que nous avons conçu s'appuie sur des algorithmes contenus dans la littérature que nous présenterons ci-après.

F. Articles intéressants de la littérature

Il existe un nombre considérable d'articles dans lesquels le concept de clique est utilisé. De plus, comme nous l'avons fait remarqué précédemment, un ensemble de sommets indépendant maximal ("independent set" ou "stable set") dans un graphe, correspond à une clique dans le graphe complémentaire. C'est pourquoi, nous avons également pris en compte les articles se rapportant aux ensembles indépendants maximaux.

Voici, pour les personnes intéressées, les références des articles que nous avons trouvé dans la littérature, ils concernent :

- la recherche de toutes les cliques dans un graphe (réf. 6, 15),
- la recherche de tous les ensembles de sommets indépendants maximaux (réf. 35, *21*, 16),
- le problème de clique proprement dit (recherche d'une clique maximum dans un graphe) (réf. 4, *14, 2*),
- la recherche d'un ensemble de sommets indépendant maximum (réf.34, 32, *14*, 34, *28, 2, 27*, 7),
- la recherche de toutes les cliques maximum dans un graphe (réf. 3, *3),*
- la recherche de tous les ensembles de sommets indépendants maximums (réf. *27, 22, 30*),
- des problèmes se rapportant aux graphes en général et adaptés aux cliques (qui sont des graphes particuliers : des graphes complets). Nous pouvons citer, par exemple, pour cette dernière catégorie, le problème des cliques dominantes dans les graphes (réf. 19, 23,

36, 10), ou le problème de couverture des cliques d'un graphe avec des sommets (réf. 12).

Nous ne tiendrons pas compte de cette dernière catégorie d'articles car ils s'écartent de notre sujet.

D'autre part, le problème de la recherche d'une clique de cardinalité la plus grande ou de toutes les cliques (cliques maximum) dans un graphe, ainsi que le problème de la recherche d'un ou de tous les ensembles de sommets indépendants maximums étant NP-complets, les travaux rapportés dans la littérature s'orientent selon deux directions : soit la recherche d'algorithmes qui solutionnent le problème pour des graphes arbitraires en un temps raisonnable mais exponentiel, soit la recherche d'algorithmes adaptés à des classes spéciales de graphes pour lesquelles des méthodes polynomiales peuvent parfois être trouvées. Les références données en *italique* ci-dessus se rapportent aux articles concernant les *graphes spéciaux*. Nous ne tiendrons pas compte non plus de ces articles, car ils nous semblent sortir du cadre du problème. En fait, nous pensons que celui-ci n'appartient pas à une catégorie bien définie de graphes. De plus, il existe de nombreuses classes différentes et nous ne connaissons généralement pas à l'avance le type de classe auquel appartient le graphe. Il est parfois possible de déterminer la classe par l'emploi de tests longs. Néanmoins, ces méthodes annuleraient les gains qu'un algorithme adapté nous donnerait. Ceci justifie notre intérêt pour les méthodes générales.

A l'exception de l'article de C. Bron et J. Kerbosch, qui est un article auquel nous faisons souvent référence, nous n'aborderons pas les articles concernant la recherche de toutes les cliques (ensembles de sommets indépendants maximaux). Notre problème, en effet, ne nécessite pas la génération de toutes les cliques mais seulement des cliques maximums. Nous nous limiterons donc aux **problèmes de recherche des cliques (ou d'ensembles indépendants) maximums dans un graphe**.

Nous proposons un très petit éventail des articles trouvés dans la littérature. Ces articles ont été choisis car nous les estimons essentiels à nos recherches. Nous nous sommes inspirés de ces articles pour concevoir un algorithme de recherche de cliques maximums adapté à notre problème.

1. C. Bron et J. Kerbosch (1973)

C. Bron et J. Kerbosch sont, en quelque sorte, des pionniers en la matière. De nombreux ouvrages, articles et applications s'appuient sur leur algorithme, en y apportant parfois des modifications appropriées à des problèmes particuliers.

La définition qu'ils utilisent pour le terme clique est : "Une clique est un sous-graphe complet maximal". Ils s'intéressent également à la recherche de **toutes** les **cliques** d'un graphe.

C. Bron et J. Kerbosch présentent deux algorithmes "**backtracking**" utilisant une technique "**branch-and-bound**" pour supprimer les branches qui ne conduisent pas à une clique. Celle-ci est expliquée et illustrée au point IV.F.1. La première version génère les cliques dans l'**ordre alphabétique (lexicographique)**. La deuxième version, quant à elle, est dérivée de la première et génère les cliques dans un **ordre imprévisible** afin de minimiser le nombre de branches qui doivent être traversées. Cette version a tendance à produire, en premier lieu, les cliques les plus grandes d'abord, puis de générer ensuite, de manière séquentielle, les cliques qui ont une grande intersection commune avec les cliques les plus grandes. Dans l'algorithme, trois ensembles jouent un rôle important :

- l'ensemble COMPSUB, est un ensemble qui doit être étendu par un nouveau point ou réduit d'un point en voyageant le long d'une branche de l'arbre de "backtracking". Les différentes configurations de COMPSUB correspondent aux noeuds dans l'arbre de "backtracking" qui comme nous l'avons vu, s'accorde à des sous-graphes complets qu'on essaie d'agrandir autant que possible dans le but d'obtenir une clique. Les points qui peuvent être choisis pour étendre COMPSUB, sont collectés de manière récursive dans les deux autres ensembles.
- L'ensemble CANDIDATES, est l'ensemble de tous les points qui serviront en temps voulu à l'extension pour la configuration présente de COMPSUB. Cet ensemble contient tous les sommets adjacents qui peuvent être ajoutés à COMPSUB. C'est-à-dire l'ensemble de tous les sommets adjacents à tous les sommets de COMPSUB, et qui n'ont pas encore servi pour étendre COMPSUB.
- L'ensemble NOT, est l'ensemble de tous les points qui ont, à une étape précédente, servi comme extension à la configuration actuelle de COMPSUB et qui sont maintenant explicitement exclus.

L'union des ensembles CANDIDATES et NOT forme l'ensemble de tous les sommets qui sont adjacents à tous les sommets de COMPSUB.

Le coeur de l'algorithme consiste en un opérateur d'extension défini **de manière récursive** qui sera appliqué aux trois ensembles décrits. Il doit générer toutes les extensions de la configuration de COMPSUB qu'il peut réaliser avec les ensembles donnés de candidats, et qui ne contiennent aucun des points de NOT. En effet, toutes les extensions de COMPSUB qui contiendraient un point de l'ensemble NOT ont déjà été générées.

Le mécanisme de base de l'algorithme comporte les **cinq étapes** suivantes :

(1) sélection d'un candidat
(2) addition de ce candidat sélectionné à COMPSUB
(3) création des nouveaux ensembles CANDIDATES et NOT à partir des anciens ensembles en supprimant tous les points non connectés au candidat sélectionné (pour rester cohérent avec la définition), et en gardant les anciens ensembles intactes.
(4) appel de l'opérateur d'extension pour intervenir sur les ensembles qui viennent d'être formés,
(5) a. lors du retour, retrait du candidat sélectionné du COMPSUB,
    b. addition du candidat à NOT.

Une condition nécessaire, mais non suffisante, pour la création d'une clique est d'avoir l'ensemble CANDIDATES vide; sans quoi COMPSUB pourrait encore être étendu. Cette condition n'est pas suffisante, car si l'ensemble NOT n'a pas encore été vidé et, si l'on se base sur de la définition de ce même ensemble NOT, nous pouvons affirmer que la configuration présente de COMPSUB a déjà été contenue dans une autre configuration, et n'est donc pas maximale. On peut donc établir que compsub est une clique dès que les deux ensembles NOT et CANDIDATES sont vides.

Si, à une étape donnée, NOT contient un point connecté à tous les autres points de l'ensemble CANDIDATES, alors on peut prévoir qu'il ne sera jamais retiré des configurations ultérieures de NOT lors des extensions futures, sélections futures de candidats, et donc ne conduiront jamais à une clique.

La méthode "branch-and-bound" permet de détecter rapidement les branches qui n'aboutiront pas à une clique.

Implantation.

L'ensemble COMPSUB se comporte comme une pile, il est maintenu et mis à jour sous la forme d'un tableau global.

Les ensembles CANDIDATES et NOT sont passés à l'opérateur d'extension en tant que paramètres. L'opérateur déclare alors un tableau local dans lequel les nouveaux ensembles sont construits, ils seront ensuite passés comme paramètre intérieur.

Les deux ensembles sont stockés dans un tableau tridimensionnel.

index values : 1 ..... ne ..... ce .....

<———> <———————>

NOT CANDIDATES

avec les propriétés suivantes :

1. $ne \leq ce$
2. ne = ce : empty (candidates)
3. ne = 0 : empty (not)
4. ce = 0 : empty (not) and empty (candidates) = clique trouvée.

si le candidat sélectionné est à la position ne+1, alors la seconde partie de l'étape 5 (ajout du candidat à NOT) est implantée avec ne = ne+1.

Un pseudo-code pour cette première version de l'algorithme de C. Bron et J. Kerbosch, repris de l'ouvrage référencé 60, est donné ci-dessous :

Pseudo-code de la version 1

```
COMPSUB← ∅
CLIQUE(V, ∅)
procédure CLIQUE(CANDIDATES, NOT)
    if CANDIDATES ∪ NOT = ∅ then output COMPSUB, which is a clique
        else if CANDIDATES ≠ ∅ then
            begin [[explore first subtree]]
                f ← vertex in CANDIDATES
                EXPLORE(f)
                [[explore remaining subtrees, if any, not precluded by theorem 1 (voir
                théorème 1 dans "La recherche exhaustive et le problème de clique".]]
                while CANDIDATES ∩ (V - Adj(f)) ≠ ∅do
                    begin
                        v ← vertex in CANDIDATES ∩ (V-Adj(f))
                        EXPLORE(v)
                    end
                end
                [[si CANDIDATES = ∅et NOT ≠ ∅ alors le théorème 2 (voir ) dit
                qu'on ne trouvera plus de nouvelle clique]]
            end
return

procedure EXPLORE(u)
    CANDIDATES ← CANDIDATES - {u}
    COMPSUB ← COMPSUB ∪ {u}
    CLIQUE(CANDIDATES ∩ Adj(u), NOT ∩ Adj(u))
    COMPSUB ← COMPSUB - {u}
    NOT ← NOT ∪ {u}
return
```

Dans la version 1 : on utilise le candidat à la position "ne + 1" comme candidat sélectionné.

Cette stratégie ne donne jamais lieu à un avancement interne équivoque et donc, toutes les cliques sont générées dans un ordre lexicographique selon l'ordonnancement initial des candidats, lors de l'appel externe (tous les points).

Description de la deuxième version.

Ici, le candidat sélectionné ne se trouve pas en position ne + 1, mais est un candidat "bien choisi"[1] à une position "s" donnée.

Pour pouvoir accomplir l'étape 5, c'est-à-dire le retrait du candidat sélectionné de COMPSUB, et son ajout à NOT, aussi simplement que dans la première version, on interverti les éléments aux positions "s" et "ne+1". Ce changement n'affecte pas l'ensemble candidates du fait qu'il n'y a pas d'ordre implicite.

La solution affecte, cependant, l'ordre dans lequel les cliques sont éventuellement générées.

Cette condition limite est formulée de la façon suivante. Il existe un point dans l'ensemble NOT connecté à tous les points dans CANDIDATES.

On aimerait que l'existence d'un tel point survienne le plus tôt possible.

Présumons qu'à chaque point de NOT soit associé un compteur du nombre de candidats, auquel ce point n'est pas connecté.

Déplacer un candidat sélectionné dans l'ensemble NOT, ce qui a lieu après extension revient à diminuer de 1 tous les compteurs des points dans NOT, auquel est connecté ce candidat sélectionné, et à introduire pour lui un nouveau compteur.

Ce compteur est toujours diminué de 1 à la fois.

Quand un compteur atteint zéro, la condition limite est atteinte.

Fixons maintenant un point particulier dans NOT.

Si l'on continue à sélectionner des candidats non connectés à ce point fixé, le compteur de ce point fixé sera diminué de 1 à chaque répétition. Aucun autre compteur ne peut diminuer plus rapidement.

Si, lors de l'initialisation, le point fixé à la valeur de son compteur au plus bas, aucun autre compteur ne pourra atteindre zéro plus tôt, aussi longtemps que les compteurs de points nouvellement ajoutés à NOT ne peuvent être plus petits. Il découle de cela la nécessité de faire entrer dans l'opérateur le point qui conduit toujours à la valeur la plus basse du compteur après la première addition à NOT. Cette fixation se fait avec le point fixé pris soit de NOT, soit de l'ensemble original CANDIDATES. Et à partir de ce moment là, on ne garde que ce compteur, en diminuant sa valeur pour chaque sélection future, du fait que l'on ne sélectionnera que des points déconnectés. Cela revient à choisir dans CANDIDATES $\cup$ NOT, le sommet qui a le plus de sommets adjacents dans CANDIDATES.

---

1 Qu'entend-t'on par "bien choisi"? En fait, un bon choix réside en une minimisation du nombre de répétitions des étapes 1-5 à l'intérieur de l'opérateur d'extension. Or, les répétitions se terminent dès que la condition qui limite l'avancement dans une branche ("bound condition") est atteinte.

Pseudo-code de la version 1

```
COMPSUB ← ≠ ∅
CLIQUE(V, ∅
procédure CLIQUE(CANDIDATES, NOT)
    if CANDIDATES ∪ NOT = ∅ then output COMPSUB, which is a clique
        else if CANDIDATES ≠ ∅ then
            begin
                f ← vertex in CANDIDATES ∪ NOT, which maximize
    | CANDIDATES - Adj(f)|
                if f ∈ CANDIDATES then [[explore first subtree]]
                EXPLORE(f)
                [[explore remaining subtrees, if any, not precluded by theorem 1 (voir
                théorème 1 dans "La recherche exhaustive et le problème de clique".]]
                while CANDIDATES ∩ (V - Adj(f)) ≠ ∅ do
                    begin
                        v ← vertex in CANDIDATES ∩ (V-Adj(f))
                        EXPLORE(v)
                    end
                end
                [[si CANDIDATES = ∅ et NOT ≠ ∅ alors le théorème 2 (voir ) dit
                qu'on ne trouvera plus de nouvelle clique]]
            end
return

procedure EXPLORE(u)
    CANDIDATES ← CANDIDATES - {u}
    COMPSUB ← COMPSUB ∪ {u}
    CLIQUE(CANDIDATES ∩ Adj(u), NOT ∩ Adj(u))
    COMPSUB ← COMPSUB - {u}
    NOT ← NOT ∪ {u}
return
```

2. R.E. Tarjan et A. E. Trojanowski (1977)

R. E. Tarjan et A. E. Trojanowski présentent un algorithme qui trouve **un ensemble de sommets indépendants maximum** dans un graphe G = (V, N), en un temps de $0(2^{n/3})$, avec n = | V| .

Ce qui, rappelons le encore une fois, est équivalent au problème de recherche d'une **clique** de taille **maximum** dans un graphe donné.

L'algorithme peut donc manipuler des graphes, environ 3 fois plus grands que ceux qu'un algorithme naïf pourrait manipuler, et ayant un temps d'exécution de $O(p(n)2^n)$, étant donné p(n), un polynôme.

Rappelons, qu'un algorithme naïf pour un graphe G = (V, E), avec n = | V| , génère tous les sous-ensembles possibles de V et teste sa propriété de clique (ou son indépendance). Le nombre de sous-ensembles de V étant 2^n, cet algorithme peut trouver une clique maximum en un temps d'exécution de $O(p(n)2^n)$, étant donné p(n) un polynôme.

Leur algorithme utilise un schéma récursif de rebroussement et dépend d'une analyse de cas compliquée et fastidieuse à établir en détail (voir annexe).

> Le point de départ de cet algorithme est l'observation suivante :
>
> Soit $v \in V$. Notons A(v) l'ensemble des sommets adjacents à v.
>
> Alors tout ensemble de sommets indépendant maximum, soit contient v, soit ne contient pas v.

Donc n'importe quel ensemble de sommets indépendant maximum de G est, soit le singleton {x} combiné à un ensemble indépendant dans G [V-{x}-A(v)], soit un ensemble indépendant maximum dans G (V-{x}).

Cette idée peut-être étendue. Pour tout $S \subseteq V$, notons $A(S) = \cup_{v \in S} A(v)$.

Si $S \subseteq V$, alors tout ensemble maximum I dans G consiste en un ensemble indépendant $I \cap S$ dans G(S) et un ensemble indépendant maximum I-S dans G [V-S-A(I)].

Leur algorithme sélectionne un **sous-ensemble** $S \subseteq V$, trouve chaque ensemble indépendant J dans G(S), et, pour chaque J, trouve **de manière récursive** un ensemble indépendant maximum dans G [V-S-A(J)].

Cette méthode est améliorée en introduisant le concept de **dominance**. comme nous le faisons maintenant.

Si $S \subseteq V$ et I, J sont indépendants dans G(S), on dit que I domine J si, pour tout $J' \subset V-S$ tel que $J \cup J'$ est indépendant, il y a un ensemble $I' \subseteq V-S$ tel que $I \cup I'$ est un ensemble indépendant et $| I \cup I'| \geq | J \cup J'|$.

Pour tout ensemble dominé J, on ne doit pas solutionner un sous-problème puisqu'on a un ensemble indépendant au moins aussi grand en solutionnant un sous-problème pour I.

La dominance est importante, parce que dans certains cas, elle peut être confirmée rapidement.

Deux exemples utilisés fréquemment dans l'algorithme sont repris ci-dessous :

Soit $v \in V$. On suppose $S = \{v\} \cup A(v)$.

Si $w \in A(v)$, alors $\{v\}$ domine $\{w\}$ dans S, car si $I \subseteq V\text{-}S$ et $I \cup \{w\}$ est indépendant, alors $I \cup \{v\}$ est indépendant.

De façon similaire, $\{v\}$ domine $\emptyset$ dans S.

On suppose $S \subseteq V$. On suppose que I et $J = I \cup \{v\}$, sont indépendants dans G(S)

On suppose $(V\text{-}S) \cap [A(v)\text{-}A(I)] = \{w_1, w_2\}$.

Dans $S \cup \{w_1, w_2\}$, J domine à la fois $I \cup \{w_1\}$ et $I \cup \{w_2\}$.

On distingue trois possibilités :

1. $(w_1, w_2) \in E$ ou $I \cap A(\{w_1, w_2\}) \neq \emptyset$.

Alors J domine I dans S : si $I' \subseteq V\text{-}S$ et $I' \cup I$ est indépendant,

alors $| I' \cap \{w_1, w_2\}| \leq 1$.

Donc $J' = I'\text{-}\{w_1, w_2\}$ satisfait $| J' \cup J | \geq | I' \cup I |$ et $J' \cup J$ est indépendant.

2. $(w_1, w_2) \notin E$, $I \cap A(\{w_1, w_2\} = \emptyset$ et

$| V\text{-}S\text{-}A(J) \cap A (\{w_1, w_2\}) | \leq 1$

Alors I domine J dans S (et $I \cup \{w_1, w_2\}$ domine J dans $S \cup \{w_1, w_2\}$: si $J' \subseteq V\text{-}S$ et $J' \cup J$ est indépendant, alors $I' = (J' \cup \{w_1, w_2\}) - A(\{w_1, w_2\})$ satisfait à $| I' \cup I | \_ | J' \cup J |$ et $I' \cup I$ est indépendant.

3. $(w_1, w_2) \notin E$, $I \cap A(\{w_1, w_2\}) = \emptyset$ et $| (V\text{-}S\text{-}A(J) \cap A (\{w_1, w_2\}) | \geq 2$.

Dans ce cas, on n'aura pas besoin de plus d'informations pour déterminer si I domine J ou l'inverse.

En résumé, l'algorithme sélectionne un ensemble $S \subseteq V$, il détermine un ensemble d'ensembles indépendants, dominant dans S, utilisant les deux observations citées ci-dessus, et il résout de manière récursive un sous-problème pour chaque ensemble dominant.

L'algorithme en pseudo-code est donné en annexe.

3. R. Carraghan et P.M. Pardalos (1990)

R. Carraghan et P.M. Pardalos proposent un algorithme qui utilise une **énumération partielle** pour résoudre le problème de **clique maximum**.

Le problème de clique, rappelons-le, est de rechercher une clique de cardinalité la plus grande.

Leur algorithme est un algorithme de "**backtracking**". Les noeuds de l'arbre sont des sous-graphes complets, que l'on essaie d'étendre pour former une clique.

R. Carraghan et P.M. Pardalos utilisent une **règle heuristique**, ainsi qu'une **méthode "branch-and-bound"**, ceci permet d'éviter de descendre le long d'une branche qui ne conduirait pas à une solution.

La règle heuristique qu'ils utilisent est un ordonnancement particulier des sommets, qui est fonction de leur degré dans le graphe. Initialement, l'algorithme considère un ordonnancement des sommets de G, disons $v_1, v_2, \ldots, v_n$ où v_1 est un sommet de plus petit degré dans $G - \{v_1\}$, et où, généralement, v_k est un sommet de plus petit degré dans $G - \{v_1, \ldots, v_k - 1\}$, $k \leq n - 2$.

Un tel ordonnancement, semble-t-il, réduit le temps calcul lorsque le graphe est dense. Alors qu'un graphe peu dense requière, pour une question de rapidité d'exécution, un algorithme sans ordonnancement.

La condition limite ("bound condition") est de regarder s'il y a encore assez de sommets pour former une clique plus grande que la plus grande clique trouvée jusqu'à présent. Sa formulation est développée plus bas.

La condition limite et la règle heuristique d'ordonnancement des sommets permettent de réduire fortement l'espace de recherche.

Un point crucial pour la compréhension de l'algorithme est la notion de **profondeur**.

A la profondeur 1, on considère tous les sommets, ordonnancés ou non, soient $v_{11} \ldots v_{1n}$.

où v_{di} est le sommet qui est étendu à la profondeur d, à l'étape i. C'est-à-dire qu'on traite le $i^{ème}$ sommet dans l'ordonnancement donné à la profondeur d donnée.

Pour la profondeur 2, on considère tous les sommets adjacents au sommet v_{1i}

A la profondeur 3, on ne maintient comme nouvel ensemble pour la profondeur *3* que les sommets de l'ensemble formé à la profondeur *2*, qui sont adjacents au sommet que l'on a choisi d'étendre à la profondeur *2*, et qui sont d'ordre supérieur dans l'ordonnancement initial.

On peut généraliser cette règle :

A la profondeur d, on ne conserve comme nouvel ensemble pour cette profondeur que les sommets de l'ensemble formé à la profondeur précédente $d-1$, qui sont adjacents au sommet

qu'on a choisi d'étendre à la profondeur *d-1*, et qui sont d'ordre supérieur dans l'ordonnancement initial. L'ensemble ainsi formé correspond à l'intersection de tous les ensembles des sommets adjacents formés aux profondeurs précédentes.

Dès lors, à chaque étape *d*, on a une suite de sommets v_{d1}, v_{di}, ..., v_{dm}, formée successivement par réduction de l'ensemble des sommets du graphe initial à chaque fois qu'un nouveau sommet est ajouté. Ainsi, seuls les sommets qui sont adjacents à tous les sommets sélectionnés jusqu'à présent, sont conservés.

En tenant compte de cette notion de profondeur, la condition limite s'exprime de la façon suivante :

Soit, la profondeur *d*, avec l'ordonnancement des sommets v_{d1}, ..., v_{di}, ..., v_{dm}; où m représente le nombre de sommet à cette profondeur et *i* indique avec quel sommet on essaie d'étendre la clique en formation.

La condition limite est que, Si $d + (m - i)$ est inférieure ou égale à la meilleure clique trouvée jusqu'à présent ("current best clique"; *CBC*), alors il faut élaguer, car la taille de la clique la plus grande possible (formée en étendant vd_i), serait inférieure ou égale à la *CBC*. Ce n'est qu'une fois à la profondeur 1 et que lorsque l'inégalité est vérifiée que l'on s'arrête. Si la condition limite n'est pas vérifiée, alors l'algorithme progresse en essayant d'étendre une clique, le plus possible, pour obtenir une clique de taille maximale supérieure à la *CBC*.

Pour mieux comprendre le fonctionnement de leur algorithme, reprenons l'exemple que R. Carraghan et P.M. Pardalos donne :

Soit le graphe G = (V, E) suivant :

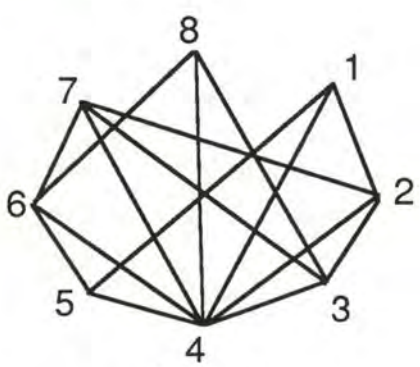


Fig. IV.4

1. l'ordonnancement initial des noeuds selon leur degré est : *1 5 8 3 2 4 6 7*
2. Profondeur 1 *: **1** 5 8 3 2 4 6 7* (le sommet en caractère gras v_{di} est étendu)
   Profondeur 2 : *5 2 4*
   Profondeur 3 : *4* (on ne sait plus étendre, la meilleure clique actuellement, ("current best clique", *CBC*) est *1, 5, 4*, de taille 3)
   Profondeur 2 : *5 **2** 4* (comme $d + (m - i) = 2 + (3 - 2) = 3 \leq 3$, on élague)

On a maintenant fini avec le noeud *1*.

La plus grande clique contenant ce noeud est *1, 5, 4*.

3. Profondeur 1 : *1 5 8 3 2 4 6 7* (le sommet *5* est étendu)
   Profondeur 2 : ***4 6*** (comme $2 + (2 - 1) \leq$ CBC, on élague)
4. Profondeur 1 : *1 5 **8** 3 2 4 6 7* (le sommet *8* est étendu)
   Profondeur 2 : ***3** 4 6*
   Profondeur 3 : ***4*** (on ne sait plus étendre; ici, on a pas formé une nouvelle meilleure clique, car la clique *8, 3, 4* $\leq CBC$)
   Profondeur 2 : *3 **4** 6* (comme $d + (m - i) = 2 + (3 - 2) = 3 \leq 3$, on élague)
5. Profondeur 1 : *1 5 8 **3** 2 4 6 7* (le sommet *8* est étendu)
   Profondeur 2 : ***2** 4 6 7*
   Profondeur 3 : ***4** 7*
   Profondeur 4 : *7* (on ne sait plus étendre; ici, *3, 2, 4, 7* devient la nouvelle meilleure clique (*CBC*), de taille 4)
   Profondeur 3 : *4 7* (comme $d + (m - i) = 3 + (2 - 2) = 3 \leq CBC$, on élague)
   Profondeur 2 : *2 **4** 6 7* (comme $d + (m - i) = 2 + (4 - 2) = 4 \leq CBC$, on élague)
6. Profondeur 1 : *1 5 8 3 **2** 4 6 7* (comme $d + (m - i) = 1 + (8 - 5) = 4 \leq CBC$ et la profondeur est 1, d'où on stoppe)

Si on est à la profondeur 1 et que la condition limite est vérifiée alors on s'arrête. On a trouvé la clique maximum.

Si on sait que la clique maximum a une taille a, alors on peut utiliser comme critère pour élaguer (stopper), le critère suivant :

$$d + (m-i) \leq a$$

Si a est proche de la taille de la vraie clique maximum, le temps de calcul peut alors être fortement réduit pour les graphes de forte densité.

Le code de cet algorithme est donné en annexe. Nous avons également adapté ce programme au problème qui nous préoccupe (voir point IV).

4. L. BABEL (1990)

L. Babel a développé un algorithme "branch-and-bound", pour trouver les cliques maximum, qui tourne rapidement sur des graphes arbitraires et qui, de plus, possède un temps d'exécution polynomial pour quelques classes spéciales de graphes.

Les idées principales développées dans leur méthode sont, **l'ordonnancement** des sommets, la **partition** du problème en sous-problèmes eux-mêmes partitionnés en sous-problèmes..., formant un **arbre de recherche** dont les noeuds correspondent aux sous-problèmes, mais aussi, la **détermination de composantes connexes** et le calcul de **limites inférieures** et **supérieures** pour la taille de la **clique maximum** de chaque composante.

Voici, de façon plus détaillée, la technique utilisée :

Schéma général de branchement et arbre de recherche exhaustive.

Étant donné un graphe G = (V, E) et un ordonnancement $(v_1, v_2, \ldots, v_n)$ des sommets, Babel définit $V_i := N(v_i) \cap \{v_{i+1} \ldots, v_n\}$ un sous-ensemble de sommets de G formé par l'intersection de l'ensemble des sommets adjacents à v_i ($N(v_i)$) dans G et du sous-ensemble des sommets de V dont le numéro d'ordre est supérieur.

Babel donne aussi le lemme suivant : si Cl est l'ensemble des sommets d'une clique dans G, alors $Cl \subseteq \{v_i\} \cup V_i$ pour au moins un $i \in \{1, 2, \ldots, n\}$. Sur base de ce lemme, le problème de clique maximum dans un graphe de cardinalité n, peut être transformé en n problèmes dans les graphes $G(V_i)$ de cardinalité d'au plus n-1, ...n.

Rappelons-le, $G(V_i)$ est le sous-graphe de G induit par V_i. Le problème $P(U_t, V_t)$ avec $U_t, V_t \subseteq V$, $Ut \cap V_t = \varnothing$ $Vt \subseteq \cap_{u \in Ut} N(u)$ qui correspond au problème MC ($G(V_t)$) : C'est-à-dire trouver une clique maximum dans G(Vt). Avec $Uo = \varnothing$ et Vo = V, le problème P(Uo, Vo) correspond au problème original MC(G). Les sommets de Ut et Vt sont appelés FIXES et LIBRES respectivement.

La structure de ces ensembles de sommets résulte du schéma général de branchement :

Etant donné $P(U_t, V_t)$ supposons $(v_1^t, v_2^t, \ldots, v_{nt}^t)$ un ordonnancement des sommets V_t.

Pour i = 1, 2, ..., n faire :

$$U_{ti} := U_t \cup \{v_i^t\}$$

$$V_{ti} := N(v_i^t) \cap \{v_{i+1}^t, \ldots, v_{nt}^t\}$$

C'est-à-dire que, pour chaque sous-problème, $P(U_t,V_t)$,caractérisé par un ensemble de sommets fixés (U_t) et un ensemble de sommets libres (V_t), on crée de nouveaux sous-problèmes, $P(U_{ti},V_{ti})$, pour chaque sommet libre (v_i^t), du sous-problème $P(U_t,V_t)$. D'une part, l'ensemble de sommets fixés (U_{ti}) d'un problème successeur est formé de l'ancien ensemble des sommets fixés (celui du problème père: U_t) auquel on ajoute le sommet libre. D'autre part, le nouvel ensemble de sommets libres est formé de l'intersection de l'ensemble des sommets adjacents au sommet libre et les sommets de l'ensemble des sommets libres du problème père (Vt) qui sont après lui.

Avec cette règle, on peut construire un arbre de recherche exhaustive dont la racine est le problème $P(U_o, V_o)$. Les sous-problèmes, générés par la règle de branchement, sont les noeuds de l'arbre.

$P(U_t, V_t)$ a $n_t = \_V_t\_$ successeurs $P(U_{t1}, V_{t1}), \ldots, P(U_{tn};V_{tn})$.

La forme de l'arbre dépendra du choix de l'ordonnancement des sommets.

Si $V_t = \varnothing$, alors il n'y a plus de branchement possible et $P(U_t, V_t)$ est une feuille de l'arbre. Sa distance à la racine est égale à $|U_t|$. Chaque feuille détenant la plus grande distance correspond à une clique maximum.

Le nombre de noeuds dans un arbre de recherche exhaustive, d'une distance au plus de k par rapport à la racine est restreint par $b(n, k) = \sum_{i=0}^{k} \binom{n}{i}$.

Et donc, pour construire l'arbre de recherche exhaustive entier, l'effort serait de

$$O\left(\sum_{i=0}^{n}\binom{n}{i}\right) = O(2^n).$$

Pour exclure l'investigation plus avancée de beaucoup de sous-problèmes, Babel introduit dans sa méthode de recherche de cliques maximum, le calcul de limites sur la taille de la plus grande clique. Cette taille est appelée le **nombre clique** (**"clique number"**) du graphe ($\omega(G)$).

Calcul de limites.

Pour calculer des **bornes** entre lesquelles est comprise la taille de la clique maximum, Babel utilise le principe de coloration.

La **coloration** d'un graphe G = (V, E) est l'application $c : V \rightarrow M$, où M est un sous-ensemble d'entiers, qui possède la propriété $(u, v) \in E \Rightarrow c(u) \neq c(v)$ pour tout $u, v \in V$. C'est donc une fonction qui est telle qu'elle associe une couleur différente à tout sommet adjacent. Si c est "surjective" et $M = \{1, 2, \ldots, k\}$ alors c est appelé une k-coloration.

Le **nombre chromatique** $\chi(G)$, est définit comme étant le plus petit k tel qu'une **k-coloration** existe pour G. Chaque $\chi(G)$-coloration est une coloration optimale. Dans un graphe coloré, tous les sommets d'une clique maximum doivent avoir une couleur différente, d'où l'inégalité : $\omega(G) \leq \chi(G)$ avec, rappelons le, $\omega(G)$ le nombre de cliques, c'est-à-dire la taille de la plus grande clique de G.

Le problème de coloration étant NP-complet, de même que le problème de clique, Babel utilise une **heuristique** de travail. Elle consiste à colorer les sommets du graphe séquentiellement avec le plus petit nombre de couleurs différentes.

L'ordre est établi en choisissant à chaque étape un sommet avec un **degré de saturation** maximal (le degré de saturation est défini comme étant le nombre de couleurs).

Cette méthode peut être étendue, sans en augmenter la complexité, pour trouver les **composantes connexes** du graphe et pour calculer simultanément les **limites inférieures** et **supérieures**, correspondant au "nombre clique" de chaque composant.

La spécification informatique de cette procédure est donnée ci-dessous :

- cdeg (v) représente le nombre de voisins colorés de v.
- satdeg (v) représente le degré de saturation de v.
- neighbcol (v) représente un ensemble contenant les couleurs qui apparaissent dans le voisinage du sommet v.

  satdeg (v) = $|$neighbcol (v)$|$
- W est l'ensemble de tous les sommets non colorés.

Procédure Bounds 1 (G).

1. For $v \in V$ do

   $c(v) := cdeg(v) := satdeg(v) := 0$, $neighbcol(v) := \varnothing$

   $W := V$, $k := 1$

   $V^1 := Cl^1 := \varnothing$, clmax := false

2. $X := \{v \in W \,|\, satdeg(v)| \geq satdeg(w)$ for all $w \in W\}$

(*) Choose $v^* \in X$ with $cdeg(v^*) \geq cdeg(w)$ for all $w \in X$

If $satdeg(v^*) = 0$ and $V \neq W$

then $(G(V^k)) := \max\{i \in N \_ \$ \, v \in V^k$ with $c(v) = i\}$

$(G(V^k)) := |Cl^k|$

$k := k + 1$

$V^k := \varnothing$, $Cl^k := \{v^*\}$, clmax := false

else if clmax = false

then if $satdeg(v^*) = |Cl^k|$

then $Cl^k := Cl^k \cup \{v^*\}$

else clmax := true

3. $c(v^*) := \min\{i \in N \,|\, i \in neighbcol(v)\}$

   $W := W - \{v^*\}$, $V^k := V^k \cup \{v^*\}$

   For $v \in N(v^*) \cap W$ do

   $cdeg(v) := cdeg(v) + 1$

   if $c(v^*) \notin neighbcol(v)$

   then $neighbcol(v) := neighbcol(v) \cup \{c(v^*)\}$

satdeg(v) := satdeg(v) +1

4. If $W \neq \emptyset$ then goto 2.

5. $\tilde{\chi}(G(V^k)) := \max \{i \in N \mid \exists v \in V^k \text{ with } c(v) = i\}$

   $\tilde{\omega}(G(V^k)) := |Cl^k|$

6. $\tilde{\chi}(G) := \max\{\tilde{\chi}(G(V^i)) \mid i \in \{1, 2, \ldots, k\}\}$

$\tilde{\omega}(G) := \tilde{\omega}(G(V^{i*})) := \max\{\tilde{\omega}(G(V^{i*})) := \max\{((G(V^i)) \mid i \in \{1, 2, \ldots, k\}\}$

   $Cl := Cl^{i*}$

Cette procédure Bounds 1, fournit pour n'importe quel graphe G, un sous-graphe complet maximal G(Cl) et une $\tilde{\chi}(G)$-coloration, et donc une limite inférieure $\tilde{\omega}(G) = |Cl|$ ainsi qu'une limite supérieure $\tilde{\chi}(G)$, pour le nombre clique ("clique number") $\omega(G)$.

Si G n'est pas connexe, alors pour chaque composante $G(V^i)$, i = 1, 2, ..., k, des limites inférieures et supérieures $\tilde{\omega}(G(V^i))$, $\tilde{\chi}(G(V^i))$ sont obtenues. Soit G un graphe non connecté avec les composants G^1, ..., G^k. A cause de la sélection d'un sommet v* qui possède le degré de saturation maximal, G est coloré composant par composant. Un nouveau composant commence si et seulement si v* a le degré de saturation 0. Pour chaque composant $G^i = G(V^i)$, des limites $\tilde{\omega}(G(v^i))$ et $\tilde{\chi}(G(v^i))$ sont calculées de la façon décrite ci-dessus. Les plus grandes limites supérieures et inférieures donnent les limites inférieures et supérieures pour $\omega(G)$.

Cette procédure Bounds 1 tourne avec un temps d'exécution de O(m+n), avec $n = |V|$ et $m = |E|$ du graphe G = (V, E).

En remplaçant la ligne (*) dans la procédure Bounds 1 par (**), Babel obtient une nouvelle procédure Bounds 2, qui donne une limite inférieure "sharper".

If clmax= false (**)

then choose $v^* \in X$ with $\deg_X(v^*) \geq \deg_X(w)$ for all $\in X$

else choose $v^* \in X$ with cdeg $(v^*) \geq$ cdeg (w) for all $\in X$

Dans le graphe partiellement coloré, X contient tous les sommets non colorés qui ont le degré de saturation le plus grand.

La procédure Bounds 1 choisit toujours un sommet $v^* \in X$, avec un degré maximal dans le sous-graphe coloré, tandis que Bounds 2 choisit, pendant la recherche de clique, le sommet v* qui est un sommet de degré maximal dans G(X), c'est-à-dire c'est le sommet qui possède le plus de sommets adjacents dans G(X).

Cette procédure modifiée commence donc en particulier avec un sommet de degré maximal dans G et peut être aussi implantée avec un temps d'exécution de l'O(m+n).

La coloration des sommets selon la taille du degré de saturation fournit des informations supplémentaires très utiles.

Les composantes connexes avec les limites inférieures et supérieures pour le "clique number" dans chacune d'elles peuvent être dérivées avec peu d'effort.

Si on considère maintenant le problème $P(U_t, V_t)$, en prenant

$G_t := G(U_t \cup V_t)$ et $G_t^i := G(U_t \cup V_t^i)$, $i = 1, 2, \ldots, k$.

Le "nombre clique" ω_t et le nombre chromatique χ_t de G_t sont donnés par

$\omega_t = |Ut| + \omega(G(V_t))$ et $\chi_t = |U_t| + \chi(G(V_t))$.

L'application de la procédure Bounds 1/2 au graphe $G(V_t)$ donne

$$\tilde{\omega}(G(V_t)) = \max \{\tilde{\omega}(G(V_t^i)) \; i \in \{1, 2, \ldots, k.\}\} \text{ et}$$

$$\tilde{\chi}(G(V_t)) = \max \{\tilde{\chi}(G(V_t^i)) \; i \in \{1, 2, \ldots, k.\}\}$$

avec les composantes connexes $G(V_t^1), \ldots, G(V_t^k))$ de $G(V_t)$.

On peut définir :

$\tilde{\omega}_t := |U_t| + \tilde{\omega}(G(V_t))$

$\tilde{\chi}_t := |U_t| + \tilde{\chi}(G(V_t))$

$\tilde{\chi}_t^i := |U_t| + \tilde{\chi}(G(V_t^i))$ avec $i = 1, 2, \ldots, k$.

$\omega_B :=$ la meilleure limite inférieure connue pour $\omega(G)$.

Si $\omega_t > \omega_B$, la limite inférieure globale peut être améliorée par $\omega_B := \omega_t$

Si $\tilde{\omega}_t \leq \omega_B$, alors on a que $\omega_t \leq \chi_t \leq \tilde{\chi}_t \leq \omega_B$, et le problème $P(U_t, V_t)$ est établi.

Le graphe correspondant G_t, ne peut contenir une clique avec plus de sommets que la plus grande que l'on aie pour le moment.

Finalement si $\tilde{\chi}_t > \omega_B$ et $\tilde{\chi}_t^i \leq \omega_B$ pour $i \in I \subseteq \{1, 2, \ldots, k\}$, le graphe G_t peut contenir une plus grande clique, alors que G_t^i, non. Donc, lorsque l'on cherche une clique maximum dans G_t nous pouvons restreindre la recherche à $G(U_t \cup V_t - \cup_{i \in I} V_t^i)$.

Règles de branchements améliorées.

Etant donné $P(U_t, V_t)$, le schéma de branchement général produit n_t sous-problèmes $P(U_{ti}, V_{ti})$.

Dans la méthode développée jusqu'à présent, chaque graphe $G(V_{ti})$ doit être coloré par la procédure Bounds 1/2. Un gain de traitement considérable peut être obtenu en utilisant l'information reçue dans la coloration de $G(V_t)$. Si $G(V_t)$ est coloré, alors chacun de ses sous-graphes $G(V_{ti})$ est coloré aussi.

Il faut simplement restreindre la coloration au sommet de ses sous-graphes.

De façon plus formelle, on peut définir $c : V \rightarrow \{1, 2, \ldots, k\}$ comme une k-coloration d'un graphe $G = (V, T)$ et $W \subseteq V$.

Alors $c : W \rightarrow \{1, 2, \ldots, k\}$ est appelé restriction de c à G(W).

On peut noter $c(W) = \{c(w) \mid w \in W\}$.

Si c est une coloration de $G(V_t)$ utilisant $(G(V_t))$ couleurs, alors $|c(V_{ti})| < \tilde{\chi}(G(V_t))$.

Evidemment, $\tilde{\chi}^{a\ priori}_{ti} = |U_{ti}| + |c(V_{ti})|$ est une limite supérieure pour ω_{ti}.

On l'appellera une **couleur limite *a priori*** pour G_{ti}.

Le schéma général de branchement marche avec n'importe quel ordonnancement des sommets.

Si on choisit l'ordonnancement intelligemment, les limites à priori, $\tilde{\chi}_{ti}^{\ a\ priori}$, peuvent être dérivées avec une facilité relativement grande.

Deux ordonnancements convenables sont les suivants :

Ordonnancement dans la séquence inverse de coloration.

Les sommets de V_t sont étiquetés dans l'ordre chronologique par $v^t_{nt}, \ldots, v^t_2, v^t_1$, c'est-à-dire :

v^t_{nt} est le premier sommet coloré v^t_1 le dernier sommet coloré. Si l'on se base sur les procédures Bounds 1/2, et plus précisément sur l'étape 2, la sélection du prochain sommet coloré est réalisée n_t fois.

Dans la $(n_{t-1} + 1)^{ième}$ itération, les sommets $v^t_{nt}, \ldots, v^t_{i+1}$ sont colorés, alors que les autres ne le sont pas.

Un sommet $v^* = v^t_i$ est choisi avec un degré de saturation maximal. Cette valeur toutefois est juste le nombre de couleurs différentes dans $G(N(v^t_i) \cap \{v^t_{i+1}, \ldots, v^t_{nt}\})$, donc

$|c(V_{ti})| = satdeg\ (v^t_i)$.

Il en résulte les règles suivantes :

Branching rule 1 :

Etant donné $P(U_t, V_t)$, avec $(v^t_{nt}, \ldots, v^t_2, v^t_1)$, un ordre de coloration des sommets V_t.

Pour $i = 1, 2, \ldots, n_t$ faire

$$U_{ti} := U_t \cup \{v_i^t\}$$

$$V_{ti} := N(v_i^t) \cap \{v_{i+1}^t, \ldots, v_{nt}^t\}$$

$$\tilde{\chi}_{ti}^{a\ priori} := |U_{ti}| + satdeg\ (v_i^t)$$

Ordonnancement selon les couleurs non croissantes.

On construit un ordonnancement $(v_1^t, \ldots, v_{nt}^t)$ tel que les couleurs de sommets n'augmentent pas quand on se déplace de gauche à droite.

Particulièrement v_1^t a la plus grande couleur, v_{nt}^t a la couleur 1, ou couleur de valeur minimale.

Comme la méthode de coloration utilisée ici, assigne à chaque sommet la plus petite couleur 1, 2, ..., $c(v_1^t)$-1.

Tous les voisins de cette zone de couleur, tout en n'ayant pas de couleur plus grande, apparaissent à la droite de v_1^t dans cet ordonnancement.

Cela montre que $|c(V_{ti})| = c(v_i^t)$-1.

On obtient la règle de branchement II :

Etant donné $P(U_t, V_t)$, avec c une coloration de $G(V_t)$, $(v_1^t, \ldots, v_{nt}^t)$ un ordonnancement des sommets avec $c(v_i^t) \geq c(v_j^t)$, $1 \leq i < j \leq n_t$.

Pour $i = 1, 2, \ldots, n_t$ faire

$$U_{ti} := U_t \cup \{v_i^t\}$$

$$V_{ti} := N(v_i^t) \cap \{v_{i+1}^t, \ldots, v_{nt}^t\}$$

$$\tilde{\chi}_{ti}^{a\ priori} := |U_{ti}| + c(v_i^t)\text{-}1$$

L'algorithme "branch-and-bound" ainsi développée par Babel est le suivant :

Algorithme BB

1. Initialisation.

   1.1. Générer la racine $P(U_t, V_t)$ de l'arbre de recherche avec $U_0 : \varnothing$ $V_0 := V$ et définir l'ensemble des noeuds actifs $AN := \varnothing$.

   1.2. Calculer les limites inférieures $\tilde{\omega}(G)$ avec la clique G(Cl), $\tilde{\omega}(G) = |Cl|$ et établir $\omega_B := \tilde{\omega}(G)$, $MCl := Cl$.

1.3. Calculer les limites supérieures $\tilde{\chi}_0 := \tilde{\chi}(G)$ pour le graphe G et $\tilde{\chi}_0^j$ pour ses composantes connexes $G(V_0^j)$, $j \in J$.

1.4. Si $\tilde{\chi}_0 \leq \omega_B$ alors aller en 5.

1.5. Pour $j \in J$ faire

Si $\tilde{\chi}_0^j \leq \omega_B$ alors $V_0 := V_0 - V_0^j$

1.6. Etablir $AN := \{P(U_0, V_0)\}$.

2. Sélection du sous-problème.

2.1. Si $AN = \varnothing$ alors aller en 5.

2.2. Choisir $P(U_t, V_t) \in AN$ selon la règle de sélection des noeuds et établir $AN := AN - \{P(U_t, V_t)\}$.

2.3. Si $\tilde{\chi}_t \leq \omega_B$ alors aller en 2.1.

3. Branchement.

3.1. Générer les successeurs $P(U_{t_1}, V_{t_1}), \ldots, P(U_{t_{n_t}}, V_{t_{n_t}})$ de $P(U_t, V_t)$ dans l'arbre de recherche selon la règle de branchement sur base des limites de couleur $\tilde{\chi}_{ti}^{\text{a priori}}$, $i = 1, 2, \ldots, n_t$.

4. Limitations.

4.1. Pour tout $P(U_{ti}, V_{ti})$ avec $\tilde{\chi}_{ti}^{\text{a priori}} > \omega_B$ faire

4.1.1. Calculer les limites inférieures $\tilde{\omega}_{ti}$ pour G_{ti} avec la clique $G(Cl)$, $\tilde{\omega}_{ti} = \lfloor Cl \rfloor$.

Si $\omega_B < \tilde{\omega}_{ti}$ alors établir $\omega_B := \tilde{\omega}_{ti}$ et $MCl := Cl$.

4.1.2. Etablir $G(V_t^j)$, $j \in J$ comme une composante connexe de $G(V_t)$.

Calculer les limites supérieures $\tilde{\chi}_{ti}$ pour le graphe G_{ti}

et $\tilde{\chi}_{ti}^j$ pour G_{ti}^j, $j \in J$.

Etablir $\tilde{\chi}_{ti} := \min\{\tilde{\chi}_{ti}, \tilde{\chi}_{ti}^{\text{a priori}}\}$

4.1.3. Si $\tilde{\chi}_{ti} \leq \omega_B$ alors aller en 4.1.6.

4.1.4. Pour $j \in J$ faire

Si $\tilde{\chi}_{ti}^j \leq \omega_B$ alors $V_{ti} := V_{ti} - V_{ti}^j$

4.1.5. Etablir $AN := AN \cup \{P(U_{ti}, V_{ti})\}$

4.1.6.

4.2. Aller en 2.

5. Solution optimale.

STOP; ω_B est le nombre clique, G(MCl) la clique maximum.

Remarque :

Les règles de sélection des noeuds établissent la stratégie qui détermine quel prochain noeud de l'arbre de recherche est traité. Par exemple, une des règles les plus actuelles, se base d'abord sur la recherche en profondeur, et ensuite sur la recherche de la meilleure limite.

Expérimentalement, la première méthode de recherche a des avantages concernant l'espace utilisé, la seconde concernant le temps d'exécution.

Si la recherche de la meilleure limite est prise en compte, alors l'étape 2.3. peut être modifiée comme critère de STOP.

2.3' If $_t \leq _B$ alors aller en 5.

Il faut modifier directement l'algorithme de telle façon que toutes les cliques maximum soient recherchées.

V. Recherche d'un algorithme efficace

Nous allons présenter le travail de développement et de test d'un algorithme "efficace" de recherche de cliques.

Le but de notre mémoire était de trouver un algorithme adéquat pour solutionner un problème rencontré dans une méthode d'alignement de séquences de protéines : "la recheche de tous les "matchs" complets.

Notre soucis étant la mise en oeuvre d'un algorithme de qualité, notre travail a consisté, d'une part en l'implantation d'un algorithme original, inspiré par les travaux de la littérature et d'autre part, en sa comparaison avec un algorithme implanté de recherche de cliques dont disposait le département de biologie quantitative. Finalement nous l'avons également comparé à un algorithme extrait de la littérature, que nous avons un peu modifié pour qu'il réponde à notre problème.

A. Démarche suivie

Etant donné le peu d'information dont nous disposions concernant les cliques, notre tâche fut, sur base bibliographique, d'extraire les informations qui retenaient notre attention.

Historiquement, la corrélation entre le problème de la recherche de tous les "matchs" complets (voir la description de la méthode d'alignement donnée au point II.B.), et le problème des cliques, a été mise en évidence lors du développement d'une méthodologie d'alignement de séquences de protéines.

Nous disposions également, au département de biologie quantitative, d'un programme écrit par Judy Hempel, en Fortran, sans spécifications, avec très peu de commentaires et très peu d'explications.

L'idée de départ de notre mémoire était,

- soit de se baser sur le programme de Judy Hempel pour obtenir un algorithme qui trouverait le plus grand nombre de cliques maximum en un minimum de temps.

  Cela supposait d'analyser plus en détails le code de ce programme pour mieux le comprendre (quitte à garder certaine partie du code comme des boîtes noires), d'y apporter des améliorations au point de vue de sa rapidité d'exécution et aussi du point de vue de la génération des cliques, et ensuite, de tester le programme sur des matrices générées avec une densité donnée de 1.

- ou, si cela était possible, de trouver un algorithme dans la littérature, clairement expliqué et efficace.

1. Etude d'un algorithme existant

Le programme de Judy Hempel a été écrit dans le cadre d'une application du problème de clique liée à l'analyse conformationnelle des protéines.

Il génère des cliques de toutes les tailles. Mais très souvent, du fait qu'il ne "parcourt" pas toutes la matrice, il ne les génère pas toutes. La recherche des cliques par cet algorithme est "chemin-dépendente". En effet, pour éviter, d'après les explications que Judy Hempel donne, de générer plusieurs fois une même clique, l'algorithme ne considère plus certains sommets comme début d'une clique.

En procédant de la sorte, un nombre considérable de cliques ne sont pas prises en compte. Ce qui permet d'accélérer les choses lorsque le nombre de clique à trouver est important. Ce point sera développé par la suite.

Une première idée d'amélioration des résultats obtenus par cet algorithme, était d'augmenter le nombre de cliques qu'il trouvait en faisant tourner plusieurs fois le programme sur la même matrice, mais dont les colonnes et les lignes étaient permutées aléatoirement. Le chemin de visite de la matrice étant différent, cela génére des cliques déjà détectées auparavant; mais également, de nouvelles cliques. Finallement, en refaisant tourner plusieurs fois l'algorithme, nous arrivons vers un plateau de saturation; c'est-à-dire qu'il n'y a pratiquement plus de nouvelles cliques trouvées.

Nous avons laissé tombé cette idée. Parce que, s'il est vrai que l'algorithme de J. Hempel est plus rapide, comme nous le verrons, qu'un algorithme qui trouve directemment toutes les cliques, il nous semble que faire tourner plusieurs fois l'algorithme sur la même matrice, dont les lignes et les colonnes sont permutées serait une perte de temps. En effet, le "gain" de temps à l'exécution est dû au fait que l'algorithme ne génère pas toutes les cliques et nous perdrions beaucoup de temps à trouver d'anciennes cliques pour essayer d'en retrouver encore de nouvelles.

Une deuxième idée était celle d'améliorer directement le code de l'algorithme pour l'optimiser et de l'adapter à notre problème. Notre problème, comme nous l'avons déjà souligné, ne concerne pas la recherche de toutes les cliques mais seulement des cliques d'une taille maximum donnée.

Sur base d'une analyse détaillée, il s'est avéré que le programme de Judy Hempel n'était pas du tout optimisé. Or, nous l'avons vu, un alignement de séquences de protéines nécessite le traitement d'un nombre considérable de graphes, chacun correspondant à une fenêtre initiale. Il est donc impératif que l'algorithme soit efficace, d'une part et valablement mis en oeuvre, d'autre part.

Comme d'un côté, le code du programme de Judy Hempel était très difficile à "débroussailler" (code inutile, code en commentaire) et qu'il ne nous était pas possible de la contacter, et comme d'un autre côté, trop de changements et de vérifications étaient à faire, nous avons également abandonnée cette idée pour définitivement nous orienter vers la littérature et les propositions que l'on y trouve.

2. Idées importantes de la littérature

Les idées importantes de la littérature que nous avons retenues, sont données ci-dessous.

- La recherche de toutes les cliques maximums dans un graphe est un problème NP-complet. Aucun algorithme, asymptotiquement efficace, n'a encore jamais été trouvé pour résoudre ce genre de problèmes.
- Plusieurs approches peuvent être envisagées face à un problème NP-complet. Ou bien il n'est pas traité dans toute sa généralité; un cas spécial du problème est traité. C'est le cas lorsque le problème de clique est résolu pour des classes spéciales de graphes. Ou bien, s'il est traité en entier, des variantes intelligentes de recherche exhaustive, comme la programmation dynamique ou la technique du "branch-and-bound" sont utilisées. Parfois, aussi, une analyse montre que les occurrences qui poseraient problème, comme par exemples celles pour lesquelles la taille des entrées serait élevé, sont rares. Dans le cas du problème des cliques, il faut analyser si la taille, ainsi que la densité des matrices que nous aurons à traiter sont souvent très grandes. Parfois encore, il n'est pas nécessaire de donner une solution exacte au problème. Dans le cas du problème de clique, nous pouvons nous contenter par exemple de rechercher le plus grand nombre de cliques et pas toutes. En fin de compte, nous avons vu que des heuristiques pouvaient être utilisées.
- Le concept de clique dans un graphe G = (V,E), est équivalent au concept d'ensemble de sommets indépendant maximal dans le graphe complémentaire.
- La technique du recherche avec rebroussement ("bactracking") combiné à une méthode "branch-and-bound" est généralement rencontrée pour résoudre le problème de clique, ou le problème des ensembles de sommets maximaux, sous toutes leurs formes (recheche de toutes les cliques, de tous les ensembles de sommets indépendant, d'une clique maximum, de toutes les cliques maximums, ...).
- Les conditions limites généralement rencontrées dans les méthodes "branch-and-bound" développées par différents auteurs sont :
    - il ne reste plus de sommet adjacent non traité à tous les sommets de la solution partielle;
    - il ne reste plus assez de sommets adjacents à tous les sommets de la solution partielle pour obtenir une clique de taille maximum; ou encore, il ne reste plus suffisamment de sommets non adjacents, pour obtenir un ensemble de sommets indépendant.
- Le problème de recherche de toutes les cliques maximum d'un graphe peut être partitionné en sous-problèmes eux-mêmes partitionnés en d'autres sous-problèmes, etc.

- Le problème de recherche de toutes les cliques maximum d'un graphe peut être partitionné en sous-problèmes eux-mêmes partitionnés en d'autres sous-problèmes, etc.
- L'introduction du concept de dominance améliore la méthode récursive de recherche avec rebroussement.

  Si un ensemble de sommets $S \subseteq V$ pour un graphe $G=(V, E)$ et que I, J sont indépendants dans G(S), on dit que I domine J si, pour tout $J' \subseteq V\text{-}S$ tel que $J \cup J'$ est indépendant, il y a un ensemble $I' \subseteq V\text{-}S$ tel que $I \cup I'$ est un ensemble indépendant et $|I \cup I'| \leq |J \cup J'|$. Pour tout ensemble dominé J, on ne doit pas solutionner un sous-problème puisque l'on a un ensemble indépendant au moins aussi grand en solutionnant un sous-problème pour I.
- l'utilisation d'heuristiques telles que l'ordonnancement des sommets selon leur degré.

3. Particularités de notre problème

Notre problème jouit de certaines particularités que nous n'avons pas rencontrées dans la littérature.

Premièrement, nous connaissons la taille de la, ou des cliques maximums qui peuvent être trouvées dans le graphe (contrairement à L. Babel et R. Carraghan et P.M. Pardalos). En effet, les segments d'une même séquence ne peuvent jamais faire parties d'une même clique. Par conséquent, une clique aura une **taille** au **maximum** égale au **nombre de groupes de "matchs"**. Ce nombre est égal au nombre de séquences que nous voulons aligner moins 1 car la séquence de laquelle est issue la fenêtre initiale, n'est pas prise en considération pour former le graphe (voir $$$).

Deuxièmement, nous recherchons toutes les cliques exactement de cette taille. C'est-à-dire que les cliques recherchées doivent comprendre toutes un "match" **issu** de **chaque** séquence (de chaque **groupe** de "matchs"). La **taille** des cliques que nous recherchons est **exactement égale au nombre de groupes.**

Nous sommes donc confrontés à un problème particulier du problème de cliques que nous pourrions qualifier de RECHERCHE DE CLIQUES MAXIMUM EN POPULATION GROUPÉE.

Dans notre algorithme, comme nous allons le voir, nous avons exploité cette caractéristique pour tenter d'augmenter sa rapidité d'exécution.

4. Idées reprises de la littérature

Pour la conception de notre algorithme, nous nous sommes fortement inspirés des algorithmes proposés dans la littérature et principalement des algorithmes de R. Carraghan et P.M. Pardalos, ainsi que de celui de L. Babel.

Notre algorithme se base également sur une **procédure récursive** de "**backtracking**".

Nous utilisons dès lors, un **arbre de recherche exhaustive** où chaque noeud correspond à une solution partielle (une clique en extension) qui doit être étendue pour trouver une solution complète (une clique maximum).

Notre algorithme repose aussi sur l'utilisation d'une méthode "**branch-and-bound**". Toutefois, comme nous l'expliquerons après, elle est différente de toutes celles que nous avons rencontrées jusqu'à présent dans la littérature.

Comme nous l'avons mentionné au point IV.D.2. concernant la recherche exhaustive, il est nécessaire d'élaguer l'arbre de recherche exhaustive pour permettre de traiter des problèmes de grande taille.

Une technique d'**élagage**, est d'empêcher les **symétries**, c'est-à-dire d'empêcher qu'une même solution soit générée plusieurs fois.

Pour cela, C. Bron et J. Kerbosch utilisent deux ensembles de sommets adjacents à tous les sommets d'une solution partielle à chaque profondeur de l'arbre de "backtracking". Un ensemble correspond aux sommets qui ont déjà été utilisés, tandis qu'un second ensemble correspond aux sommets qui peuvent servir à l'extension d'une nouvelle clique. R. Carraghan et P.M. Pardalos, ainsi que L. Babel eux, n'utilisent qu'un seul ensemble de sommets dans lequel sont conservés seulement les sommets adjacents à tous les sommets de la solution partielle, qui sont d'ordre supérieur à ceux-ci, dans l'ordonnancement initial des sommets. De même, nous n'utiliserons qu'un seul ensemble.

Une autre technique d'**élagage** est d'introduire une **condition limite** (technique "branch-and-bound") permettant d'arrêter les recherches dans une branche de l'arbre, dès que l'on sait qu'une solution ne peut être atteinte.

Dans l'algorithme de R. Carraghan et P.M. Pardalos, la recherche est arrêtée, dès qu'on sait que la nouvelle clique ne sera pas de taille supérieure ou égale à la meilleure clique trouvée jusqu'alors.

De par la particularité de notre problème, nous obtenons un critère d'arrêt supplémentaire. Cette particularité, rappelons-le encore une fois, est que les cliques recherchées doivent contenir exactement un "match", issu de chacune des séquences comparées avec la séquence dont provient la fenêtre initiale. En conséquence, elles ne peuvent avoir une taille inférieure, ni supérieure au nombre de groupes de "matchs". Cette remarque débouche sur le fait que nous obtenons en fait le critère d'arrêt supplémentaire suivant :

"Dans une des séquences comparées à la fenêtre initiale, nous pouvons arrêter la recherche dès qu'il n'y a plus de "matchs" adjacent à tous les "matchs" contenus dans la solution partielle".

Cela signifie qu'une clique maximum de taille égale au nombre de groupes ne sera jamais trouvée dans cette branche de l'arbre.

Ce critère d'arrêt nous semble particulièrement intéressant car, il nous permet parfois de couper des branches haut dans l'arbre de recherche.

Comme Babel, nous pouvons exprimer notre problème en terme de partition en sous-problèmes :

- Le problème $P(C_i,V_i)$ correspond au problème MC(Gi): "trouver toutes les cliques maximums du graphe G (Vi)". C_i correspond à un ensemble de sommets qui forment un graphe complet. Autrement dit, C_i est une solution partielle, une clique de taille i, destinée à être étendue pour former une clique maximum. $V_i \subseteq V$ et plus précisément $V_i \subseteq \bigcap_{c \in C_i} N(c)$.
- Le problème initial $P(C_0,V_0)$ correspond au problème original MC(G): "trouver toutes les cliques maximums de G = (V,E). $V_0 = V$ et $C_0 = \varnothing$".
- L'arbre de recherche a comme racine P(C0,V0) et les noeuds de l'arbre sont :

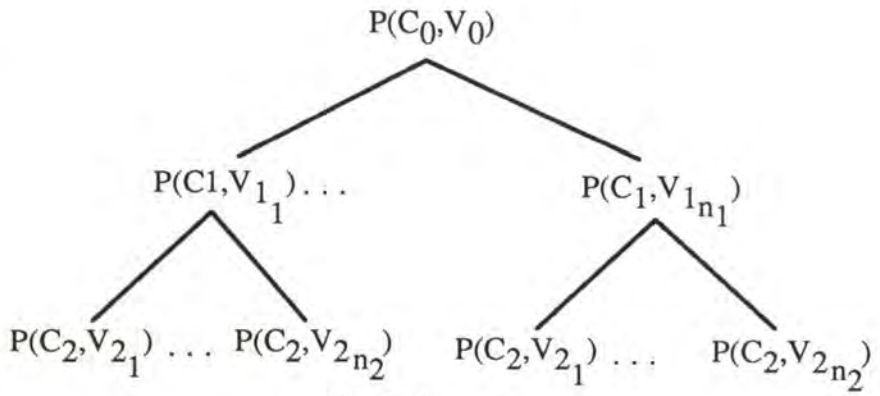


Fig. V.1

- La profondeur de l'arbre de recherche est égale au nombre de groupes de "matchs", qui est aussi la taille d'une clique maximum.
- Les sous-problèmes créés dans l'arbre de toutes les possibilités sont définis comme suit : soit $P(C_i, V_i)$, un problème qui est décomposé. Pour chaque sommet v_j du groupe de "matchs" $i + 1$, l'ensemble de sommets $V_{i_j} := N(v_j^i) \cap v_{l_i+1}, ...,v_{l_s}$, où l_i correspond à l'indice du dernier "matchs" du groupe i et l_s, au dernier "match" du dernier groupe. $C_{i_j} := C_i \cup v_i^j$.

Cette stratégie permet de résoudre le problème de manière récursive en solutionnant des problèmes de plus petite taille. Toutefois, nous utilisons une méthode groupée et nous connaissons la taille d'une clique maximum. Cela implique que nous avons moins de sous-problèmes à traiter, au premier niveau de l'arbre. En effet, à ce niveau, nous n'avons de

sous-problèmes que pour les sommets du premier groupe et non pas pour tous les sommets du graphe comme dans la stratégie développée par L.Babel. Nous évitons ainsi le problème de symétrie qui justifiait l'utilisation de deux ensembles de sommets. Cela nous semble être un gain appréciable. De plus, nous n'avons pas besoin d'heuristiques pour estimer la taille d'une clique maximum car nous la connaissons. De ce fait nous connaissons également la profondeur de l'arbre de recherche et le nombre de noeuds maximums qu'il aura. A chaque niveau, le nombre maximum de sous-problèmes sera égal au nombre de "matchs" trouvés dans le groupe correspondant à ce niveau. D'où, le nombre total de noeud sera au maximum égal à $n_1 * ... * n_i * ... * n_s$. n_i est le nombre de "matchs" trouvés dans la séquence i et n_s est le nombre de "matchs" trouvés dans la séquence s, avec s, le nombre de groupes de "matchs".

B. Résolution du problème abstrait

Pour mieux faire comprendre les grandes lignes de notre algorithme, nous allons l'exécuter de manière abstraite sur un petit exemple :

Posons le problème :

Soit I, A, B, C, D, les séquences en acides aminés de cinq protéines que nous voulons aligner.

La procédure de "matching" de la méthode d'alignement a identifié pour la fenêtre initiale I_0, les "matchs" A_1, A_2 et A_3 dans la séquence B_1, B_2 et B_3, dans B, C_1, C_2, C_3, C_4 et C_5 dans C, et D_1, D_2 et D_3.

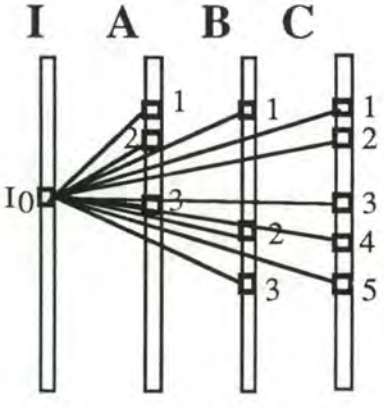


Fig. III.5.

Nous recherchons les "matchs" complets.

Pour rappel, un "match" complet est un ensemble de segments de séquences de protéines, tous issus d'une protéine différente

La représentation matricielle de ce problème est la suivante :

	A1	A2	A3	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
A1	1	0	0	1	1	0	1	0	0	1	0	0	1	0
A2	0	1	0	1	0	0	1	1	1	0	0	1	0	1
A3	0	0	1	1	0	1	1	0	1	0	1	0	0	1
B1	1	1	1	1	0	0	0	1	0	1	1	1	0	1
B2	1	0	0	0	1	0	1	0	0	1	1	1	0	1
B3	0	0	1	0	0	1	0	1	1	0	0	1	1	1
C1	1	1	1	0	1	0	1	0	0	0	0	0	1	0
C2	0	1	0	1	0	1	0	1	0	0	0	1	1	1
C3	0	1	1	0	0	1	0	0	1	0	0	0	1	0
C4	1	0	0	1	1	0	0	0	0	1	0	0	0	0
C5	0	0	1	1	1	0	0	0	0	0	1	0	1	0
D1	0	1	0	1	1	1	0	1	0	0	0	1	0	0
D2	1	0	0	1	0	1	1	1	1	0	1	0	1	0
D3	0	1	0	1	1	1	0	1	0	0	0	0	0	1

Pour la conception de notre algorithme, nous avons repris la notion de profondeur développée par R. Carraghan et P.M. Pardalos.

De la même façon que ces auteurs, nous travaillons avec des ensembles de sommets candidats, qui sont restreints successivement, pour former une clique. Lorsque nous sélectionnons un candidat, nous supprimons les sommets qui ne sont pas adjacents au sommet étendu. De façon pratique, nous mettons une valeur zéro dans un vecteur (le vecteur CANDIDAT), pour indiquer qu'ils ne peuvent plus être utilisés pour étendre la clique. Une valeur 1 indique, bien entendu, qu'ils peuvent servir pour étendre la clique en formation. Nous appellerons CLIEXT cette clique en extention.

Notre algorithme procède comme suit :

1. Initialisation :

A1	A2	A3	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
1	1	1	1	1	1	1	1	1	1	1	1	1	1

A l'initialisation, tous les "matchs", c'est-à-dire les sommets du graphe, sont considérés comme des candidats possibles pour former une clique maximum. Le vecteur CANDIDAT est, dès lors, entièrement rempli de 1.

Pour tous les "matchs" du premier groupe, nous essayons de former une clique maximum.

2. Profondeur 1 :

A1	A2	A3	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
1	1	1	1	1	1	1	1	1	1	1	1	1	1

a. On sélectionne un candidat dans le premier groupe pour étendre une clique

Rem. : le grisé indique le groupe dans lequel on doit choisir un candidat adjacent à tous les sommets formant une clique en extension. Le sommet en gras est celui qu'on essaye d'étendre.

	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
	1	1	1	1	1	1	1	1	1	1	1
A1	1	1	1	1	0	0	1	0	0	1	0
A1	1	1	1	1	0	0	1	0	0	1	0

b. On vérifie le fait qu'il existe encore au moins un candidat dans tous les groupes après l'addition du candidat sélectionné. Dans l'affirmative, la clique en formation trouvée à la profondeur précédente est étendue avec ce candidat (A1 est le début d'une clique).

Comme le tableau ci-dessus l'illustre, il suffit pour cette vérification, de faire une multiplication booléenne d'une partie du vecteur CANDIDAT avec une partie de la ligne de la matrice correspondant au candidat sélectionné. Et ensuite, il suffit de vérifier s'il y a au moins un "match" adjacent à tous les sommets de la clique en extension (CLIEXT, ici est A1). Autrement dit, si toutes les entrées du vecteur résultant sont nulles pour un groupe, alors nous pouvons élaguer. En effet, nous ne formerons jamais de clique de taille maximum égale au nombre de groupes. Tandis qu'après la multiplication, il existe encore au moins une entrée non nulle dans chaque groupe, nous passons à l'étape *c*.

c. Retrait de tous les matchs qui ne sont pas adjacents au "match" étendu à la profondeur précédente, dans le but de former l'ensemble des candidats pour la profondeur suivante, et essayer de manière récursive d'étendre la clique.

Remarque : cela consiste à prendre, comme nouvel ensemble candidat, le vecteur résultant de la multiplication.

3. Profondeur 2 :

	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
A1	1	1	1	1	0	0	1	0	0	1	0

a. On sélectionne le premier "match" du groupe des B adjacents au match sélectionné à la profondeur précédente, c'est-à-dire à A1 (=B1).

	C1	C2	C3	C4	C5	D1	D2	D3
A1	1	0	0	1	0	0	1	0
B1	0	1	0	1	1	1	0	1
A1 B1	0	0	0	1	0	0	0	0

b. On vérifie qu'il existe encore au moins un candidat dans tous les groupes après l'addition du candidat sélectionné.

Ici, le goupe des D ne contient aucune entrée non nulle, donc nous élaguons.

Nous retournons à l'étape a et nous sélectionnons le "match" suivant dans le groupe de B pour lequel la valeur de candidat vaut 1 (=B2).

	C1	C2	C3	C4	C5	D1	D2	D3
A1	1	0	0	1	0	0	1	0
B2	1	0	0	1	1	1	0	1
A1 B2	1	0	0	1	0	0	0	0

Toutes les entrées dans le groupe D sont nulles. Donc, nous élaguons et nous retournons à l'étape a. Nous sélectionnons alors le match suivant s'il existe dans le groupe B pour lequel la valeur candidat vaut 1. Comme il n'y en a plus, nous élaguons et nous retournons à la profondeur 1.

3. Profondeur 1 :

A1	**A2**	A3	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
1	1	1	1	1	1	1	1	1	1	1	1	1	1

a. On sélectionne le prochain candidat (A2) dans le premier groupe.

	A1	**A2**	A3	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
	1	1	1	1	1	1	1	1	1	1	1	1	1	1
A2	0	1	0	1	0	0	1	1	1	0	0	1	0	1
A2	0	1	0	1	0	0	1	1	1	0	0	1	0	1

b. Nous vérifions qu'il reste pour ce candidat sélectionné, au moins un "match" adjacent dans les autres groupes.

c. Pour former l'ensemble des candidats de la profondeur suivante, et essayer de manière récursive d'étendre la clique en extension obtenue à la profondeur précédente, on retire tous les matchs qui ne sont pas adjacents à tous les sommets de celle-ci.

Profondeur 2 :

	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
A2	1	0	0	1	1	1	0	0	1	0	1

a. On sélectionne le premier "match" du groupe des B adjacent à A2 (=B1).

b. On vérifie qu'il existe encore au moins un candidat dans tous les groupes après l'addition du candidat sélectionné.

		C1	C2	C3	C4	C5	D1	D2	D3
	A2	1	1	1	0	0	1	0	1
	B1	0	1	0	1	1	1	0	1
A2	B1	0	1	0	0	0	1	0	1

Remarque : Tous les groupes possèdent des candidats. A2B1 forme une clique en extension et nous pouvons passer à l'étape c.

c. On retire tous les "matchs" non adjacents au candidat sélectionné; on passe ensuite à la profondeur suivante.

Profondeur 3 :

	C1	C2	C3	C4	C5	D1	D2	D3
A2 B1	0	1	0	0	0	1	0	1

a. On sélectionne le premier "match" du groupe adjacent à tous les "matchs" de la clique en extension, c'est-à-dire, adjacent à A2 et à B1 (= C2).

b. On vérifie qu'il existe encore au moins un candidat dans tous les groupes après l'addition du candidat sélectionné.

			D1	D2	D3
A2	B1		1	0	1
		C2	1	1	1
A2	B1	C2	1	0	1

Tous les groupes possèdent des candidats. C2 est ajouté à la clique en extension. Nous pouvons passer à l'étape c.

c. Nous retirons tous les "matchs" non adjacents; et nous passons à la profondeur suivante.

Profondeur 4 :

			D1	D2	D3
A2	B1	C2	1	0	1

Ici, la profondeur est égale au nombre de groupes de "matchs". Les cliques maximums correspondent aux cliques formées des sommets sélectionnés aux profondeurs précédentes, et de chaque "match" de cette dernière profondeur adjacent à ceux-ci.

Nous avons ainsi formé les cliques maximums (A2 B1 C2 D1 et A2 B1 C2 D3) et nous retournons à la profondeur précédente.

Profondeur 3 :

		C1	C2	C3	C4	C5	D1	D2	D3
A2	B1	0	1	0	0	0	1	0	1

a. Il n'y a plus de "matchs" candidats dans le groupe des C; alors nous élaguons.

Profondeur 2 :

	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
A2	1	0	0	1	1	1	0	0	1	0	1

a. Il n'y a plus de "matchs" candidats dans le groupe des B; d'où, nous élaguons.

2. Profondeur 1 :

B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
1	1	1	1	1	1	1	1	1	1	1

a. On sélectionne le prochain candidat dans le premier groupe pour étendre une clique.

	A1	A2	**A3**	B1	B2	B3	C1	C2	C3	C4	C5	D1	D2	D3
	1	1	1	1	1	1	1	1	1	1	1	1	1	1
A3	1	0	0	1	1	1	1	0	0	1	0	0	0	0
A3	1	0	0	1	1	1	1	0	0	1	0	0	0	0

b. On vérifie qu'il existe pour ce candidat sélectionner au moins un "match" adjacent dans les autres groupes.

Ici, d'une part, le groupe D ne comprend aucune entrée non nulle et d'autre part, nous sommes au dernier sommet du premier groupe de la première profondeur. La recherche de toutes les cliques de taille maximum est terminée. Ces tailles correspondent au nombre de groupes.

Nous pouvons déduire aisément notre algorithme de la description donnée ci-dessus. Cet algorithme est décrit en pseudo-code dans le point qui suit.

C. Solution algorithmique

1. Structure de données

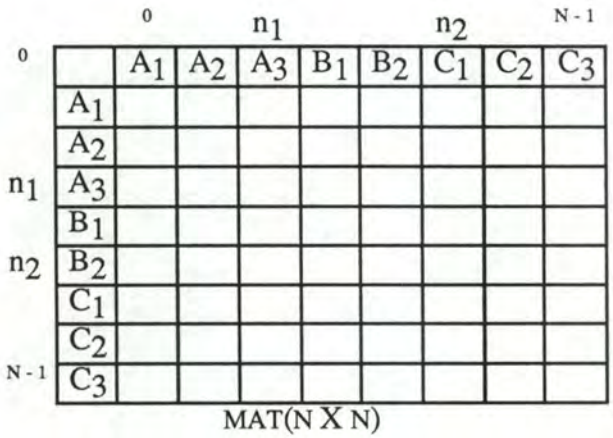
Les structures de données nécessaires au programme sont :

a. Constantes du problème

- Une **matrice d'adjacence** : MAT.

Sa taille est : N X N. N est la somme de tous les "matchs" trouvés dans tous les groupes; N correspond au nombre de sommets du graphe.

MAT(i, j) = 1 ($1 \leq i, \leq N$) si le "match" i est similaire au "match" j, et que i et j correspondent à des "matchs" issus de séquences de protéines différentes; et 0, sinon.



MAT(N X N)

- Un tableau d'indices : IFINGPE.

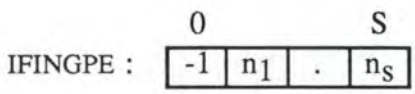
Il donne l'indice de la colonne du dernier "match" des différents groupes.

Sa taille est la suivante : 1 X (S + 1). S est le nombre de groupes, c'est-à-dire le nombre des séquences dont sont issus les "matchs".

m_i est le nombre de match trouvés dans la séquence *i* (pour le groupe *i*).

m_i = IFINGPE(i) - IFINGPE(i - 1).

N = la somme des m_i.



b. Variables globales du problème

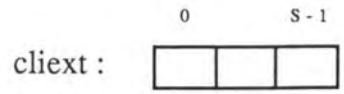
- Une variable X :

X détermine le groupe courant qui est étendu. C'est une variable qui pointe vers l'indice de la colonne correspondant au dernier "match" du groupe précédent. Cette variable aura la valeur - 1, si nous sommes au premier groupe.

$1 \leq X \leq S$

- Un vecteur CLIEXT.

La taille de CLIEXT est : (1 X S) avec S, le nombre de séquences.



CLIEXT contient une clique en extension.

- Un variable K.

K indique la position de la prochaîne entrée dans le vecteur CLIEXT.

Lors de l'initialisation K est mise à la valeur 0. Le programme est prêt à écrire le premier sommet de la clique dans CLIEXT(0).

Quand nous avons une clique maximum : K = S. Dans ce cas, tout le tableau est rempli.

c. Variables locales du problème

a. locales à FINDCLI, la procédure récursive qui recherche une clique maximum à partir de l'extension d'une clique donnée :

- Une variable J.

J indique avec quel sommet du groupe nous étendons la configuration actuelle de CLIEXT. Chaque fois que nous rentrons dans FINDCLI, cette variable est initialisée par le premier sommet du groupe suivant. Par la suite, dans la procédure FINDCLI, une boucle la fait progresser jusqu'au dernier sommet du groupe. D'où, IFINGPE(X) + 1 ≤ J ≤ IFINGPE(X + 1).

b. locales à la procédure EXISTCANDEXT, qui vérifie s'il existe encore des candidats possibles dans chacun des groupe :

- Une variable Z.

Z est initialisé par le groupe suivant celui du "match" que nous essayons d'étendre. Elle permet de passer de groupe en groupe tant qu'ils comprennent encore des candidats possibles; 2 ≤ Z ≤ S - 1.

- Une variable Y.

Y pointe vers le "match" du groupe pour lequel nous vérifions s'il sera un candidat possible lorsque le sommet que nous étendons sera rajouté à CLIEXT. IFINGPE(Z) ≤ Y ≤ INFINGPE.

- Une variable OK.

OK est une variable booléenne qui est initialisée à faux chaque fois que nous changeons de groupe et passe à vrai s'il y a un candidat possible dans ce groupe.

2. Pseudo-code du programme de recherche de cliques

a. Procédure : existcandext

Paramètres d'entrée :
candext : pointeur sur entier;
candidat : pointeur sur entier;
j : pointeur sur entier;
Valeur renvoyée de type entier.
Variables locales :
y, z, ok : entier;

```
DEBUT
  z ← x + 1;
  ok ← FAUX;
  y ← ifingpe[z] + 1;
  SI y = N ALORS
    renvoyer VRAI et sortir;
  SINON
    REPETER
      SI y <> N ET y <> ifingpe[z + 1] + 1 ALORS
        candext[y] ← mat[pointeur sur j * N + y] * candidat[y];
        ok ← ok OU candext[y];
        y ← y + 1;
      SINON
        SI ok = FAUX ALORS
          renvoyer FAUX et sortir;
        SINON
          SI y = N ALORS
            renvoyer VRAI et sortir;
          SINON
            ok ← FAUX;
            z ← z + 1;
          FIN_SI
        FIN_SI
      FIN_SI
    FIN_REPETER
  FIN_SI
FIN
```

b. *Procédure : findcli*

Paramètres d'entrée :
candidat : pointeur sur entier;
Pas de valeur renvoyée.
Variables locales :
j : entier;

```
DEBUT
  j ← ifingpe[x] + 1;
  TANT_QUE j <> N  ET j <> ifingpe[x + 1] + 1 FAIRE
    appel de : traitersommet(adresse de j, candidat);
  FIN_TANT_QUE
FIN
```

c. *Procédure : traitersommet*

Paramètres d'entrée :
j : pointeur sur entier,
candidat : entier.
Pas de valeur retournée.
Variables locales :
candext : pointeur sur entier.

```
DEBUT
  SI candidat[pointeur sur j] = 1 ALORS
    allouer mémoire à candext de dimension N;
    SI existcandext(candext, candidat, j) ALORS
      cliext[k] ← pointeur sur j;
      SI k = S - 1 ALORS
        appel de : memoriserclique;
      SINON
        x ← x + 1;
        k ← k + 1;
        appel de : findcli(candext);
        x ← x - 1;
        k ← k - 1;
      FIN_SI
    FIN_SI
    appel de : free(candext);
  FIN_SI
  pointeur sur j ← (pointeur sur j) + 1;
FIN
```

d. *Procédure : trt_clique*

Paramètres d'entrée :

candidat : pointeur sur entier;

j : pointeur sur entier;

Pas de valeur renvoyée.

Variables locales :

z : entier;

```
DEBUT
  cliext[k] ← pointeur sur j;
  k ← k + 1;
  DE z ← ifingpe[x] + 1 A z < N PAR z++ FAIRE
    candidat[z] ← mat[pointeur de j * N + z];
  FIN_DE
FIN
```

e. *Corps principal*

Pas de paramètres d'entrée.

Variables locales :

j : entier;

candidat : pointeur sur entier;

Pas de valeur renvoyée.

```
DEBUT
  SI NON  read_matrix() ALORS
    SI NON read_igroup() ALORS
      appel de : view_matrix();
      result ← ouvrir fichier RESULTAT.DAT;
      j ← 0;
      appel de : init_all();
      allouer mémoire à candidat, de dimenssion N;
      TANT_QUE j <> ifingpe[x] + 1 FAIRE
        appel de : trt_clique(candidat, adresse de : j);
        appel de : findcli(candidat);
        k ← k - 1;
        j ← j + 1;
      FIN_TANY_QUE
      appel de : fclose(result);
      appel de : free(candidat);
      appel de : free(igroup);
      appel de : free(ifingpe);
      appel de : free(cliext);
    FIN_SI
    appel de : free(mat);
    appel de : free(column);
  FIN_SI
FIN
```

VI. Test du nouvel algorithme

A. But des tests

En vue d'étudier l'efficacité de notre programme, nous l'avons testé sur des matrices générées aléatoirement avec une taille et une densité données. Nous l'avons aussi "comparé" au programme implanté, *Brandnewcli*, de J. Hempel et à celui de R. Carraghan et P.M. Pardalos.

Pour cela, nous avons implanté l'algorithme de R. Carraghan et P.M. Pardalos avec certaines modifications. Au point de départ, il avait été conçu pour donner la taille et les sommets d'une clique maximum. La condition limite qui stoppait l'extension d'une solution partielle était qu'elle n'aboutirait jamais à une clique maximum. Ce qui est le cas, lorsque le nombre de sommets adjacents à tous les sommets de la solution partielle, additionné au nombre de sommet formant la solution partielle est inférieur ou égal à la taille de la meilleure clique obtenue jusqu'alors. Par conséquent, pour obtenir toutes les cliques, nous avons éliminé l'égalité.

L'algorithme de R. Carraghan et P.M. Pardalos, utilise par ailleurs, une heuristique qui devrait permettre d'élaguer l'arbre de recherche lorsque le graphe est dense (densité ≥ 0.4). Cette heuristique consiste premièrement en un ordonnencement des sommets qui est fonction de leur degré dans le graphe initial et deuxièmement, à choisir le sommet qui possède le degré le plus élevé pour l'extension d'une clique. Toutefois, cette heuristique nous semblait surtout intéressante pour accélérer la recherche d'une seule clique maximum. Mais elle nous semblait beaucoup moins intéressante dans le cas de la recherche de toutes les cliques maximums. Aussi, dans le but d'éprouver cette hypothèse, nous avons implanté deux versions différentes de l'algorithme : une version qui réalise l'ordonnencement lorsque la densité de la matrice est ≥ 0.4, et une deuxième version qui ne réalise jamais d'ordonnencement.

Pour juger de l'efficacité du critère d'arrêt que nous avons utilisé (à savoir la présence de sommets candidats à l'extension dans tous les groupes; voir point V.A.4), nous avons aussi implanté deux versions différentes de notre algorithme : une dans laquelle le critère d'arrêt est utilisé et, une autre dans laquelle il n'est pas utilisé.

B. Organigramme des programmes utilisés pour réaliser les tests

L'organigramme ci-dessous reprend la totalité des programmes utilisé dans la procédure de tests à savoir :

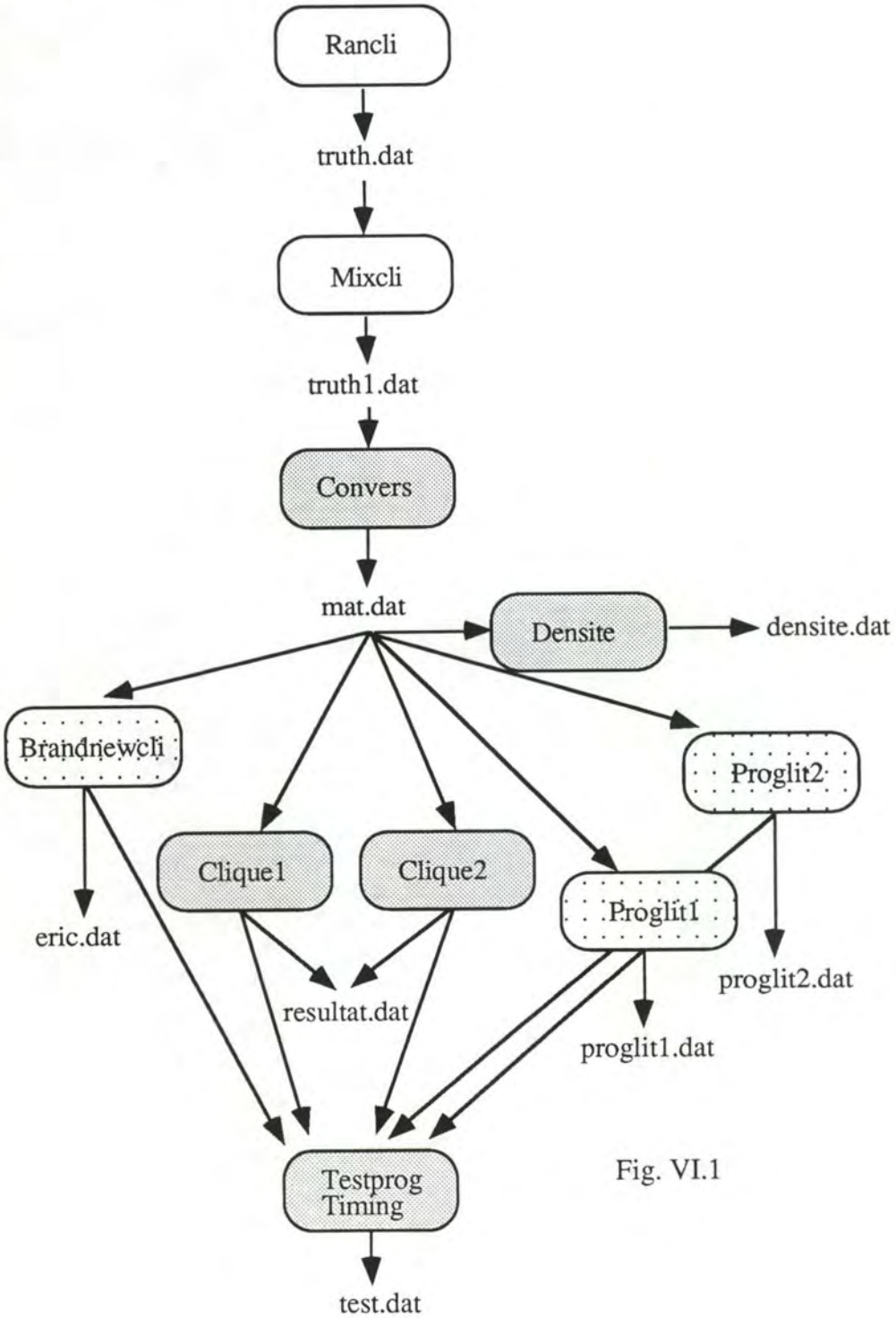


Fig. VI.1

- Rancli et Mixcli sont des programmes, qui avaient été conçus au département de biologie quantitative pour générer des matrices d'une taille et d'une densité de 1 données. Ils servaient à tester la rapidité d'exécution du programme Brandnewcli.

- Convers est un programme que nous avons écrit, qui sert à convertir la matrice originale générée par Rancli et Mixcli en une matrice qui inclut la notion de groupe de"matchs". Cela implique que la nouvelle matrice générée, "mat.dat" possède des groupes au sein desquels la similarité entre "match" est considérée comme nulle. Donc, mat(i,j) = 0 pour tout i≠j du même groupe de "matchs". C'est cette matrice que nous utiliserons pour comparer la rapidité d'exécution, et aussi les resultats fournis par les différents programmes.

- Densité est un programme que nous avons écrit et qui donne le pourcentage de 1 trouvé dans la nouvelle matrice, *mat.dat.*

- Brandnewcli est le programme écrit par Judy Hempel. Il génère un fichier résultat "eric.dat" avec toutes les cliques trouvées. Pour les tests, nous avons modifié légèrement ce programme pour qu'il n'imprime que les cliques maximums dans *eric.dat.*

- Proglit1 est le programme de R. Carraghan et P.M. Pardalos que nous avons modifié pour qu'il génère toutes les cliques maximums, sans aucun ordonnancement des sommets. Les résultats de l'exécution sont stockés dans le fichier *proglit1.dat*

- Proglit2 est la deuxième version tirée du programme de R. Carraghan et P.M. Pardalos. Ici, nous l'avons aussi modifié pour qu'il génère toutes les cliques maximums mais, nous avons conservé la partie de code qui réalise un ordonnencement des sommets lorsque la densité est ≥ 0.4. Les résultats de l'exécution sont stockés dans le fichier *proglit2.dat*

- Clique1 est la première version de notre programme. Elle génère toutes les cliques maximums dans l'ordre lexicographique avec le critère d'arrêt supplémentaire : la vérification de la présence de candidats dans tous les groupes (voir V.A.4). Les cliques maximums sont stockées dans le fichier resultat.dat.

- Clique2 est la deuxième version de notre programme. Elle génère aussi toutes les cliques maximums dans l'ordre lexicographique mais sans le critère d'arrêt supplémentaire (voir V.A.4); les cliques maximums sont stockées dans le fichier resultat.dat.

Bien entendu, nos deux versions donne exactement le même nombre de cliques et le même fichier résultat.

- Timing.com est une procédure batch qui dirige l'exécution des programmes que nous comparons (Brandnewcli, Proglit1, Proglit2, Clique1, Clique2) et donne le temps CPU consommé par chacun.

- Testprog.com est une procédure batch destinée à diriger l'introduction et le stockage des données (taille de la matrice, densité, nombre de groupes, nobre de "matchs" par groupe), la création de la matrice (c'est-à-dire, l'exécution de Rancli, Mixcli, Convers), l'exécution de la procédure Timing, le stockage des résultats donnés par Timing et finalement, l'extraction et le stockage du nombre de cliques trouvées dans chaque fichier résultat. Testprog.com produit un fichier *test.dat* reprenant tous ces éléments.

C. Choix des tests et résultats obtenus

Les programmes dont nous disposons pour rechercher les cliques maximums dans une matrice ne peuvent être vraiment comparés. Premièrement, ils n'ont pas été conçus pour les mêmes problèmes.

A l'origine, Brandnewcli, a été conçu pour générer des cliques de toutes les tailles, l'algorithme "proglit" de R. Carraghan et P.M. Pardalos, pour générer la taille et les sommets d'une clique maximum, avec ou sans ordonnancement des sommets, notre programme "clique" pour générer toutes les cliques maximums, avec ou sans critère d'arrêt.

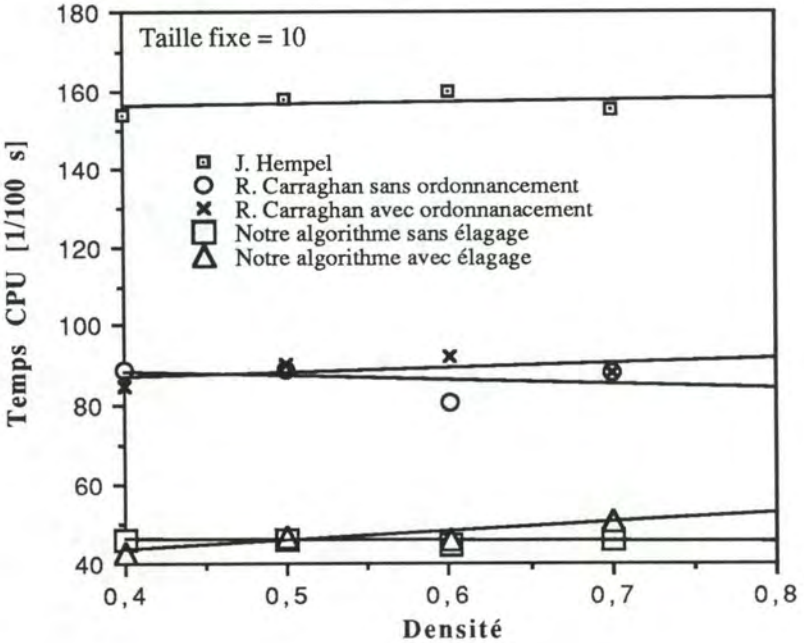
Deuxièment, ils n'ont pas tous été programmés dans le même langage. Les premiers ont été programmés en Fortran, tandis que le nôtre a été programmé en C. Nous avons choisi ce langage pour permettre l'intégration de ce programme à l'ensemble des procédures de la méthode d'alignement.

De plus, il nous est difficile de faire une analyse rigoureuse et détaillée car beaucoup de données interviennent dans notre problème. La taille de la matrice, la densité et le nombre de groupe de "matchs" vont influencé les résultats Les tests que nous avons réalisés couvrirons seulement une partie des combinaisons possibles de ces résultats.

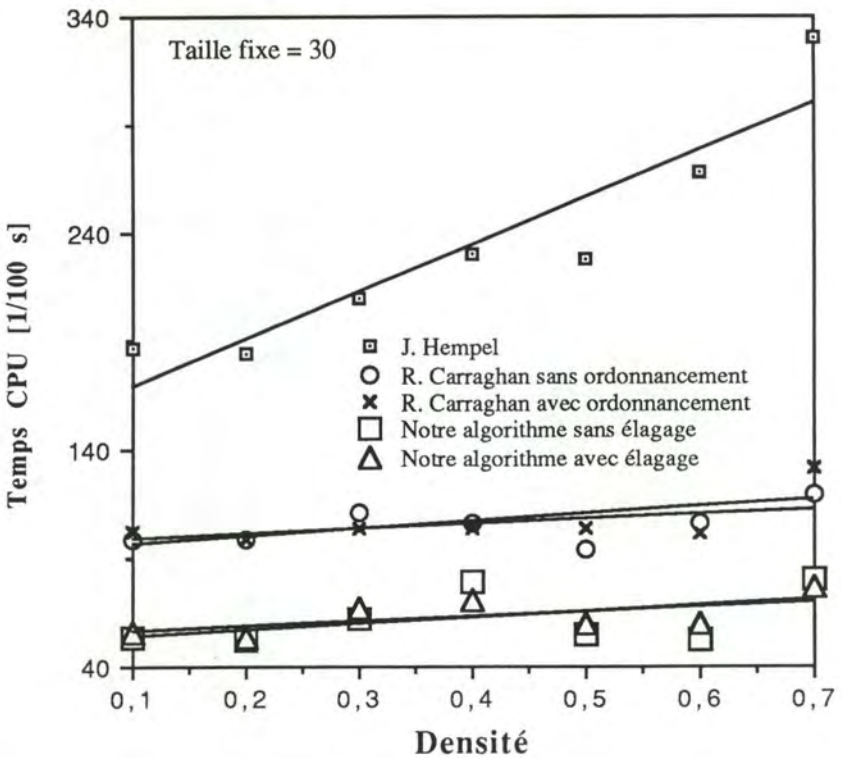
Par ailleurs, ne disposant pas de la place mémoire nécessaire, nous n'avons été limité à des tailles de matrice relativement faible. Ces tests devront donc être complétés.

Nous donnons ci-dessous des graphes représentant la variation du temps CPU consommé par chaque algorithme en fonction de la densité avec une taille fixe de 10 pour le premier graphe et une taille fixe de 30 pour le second graphe. Les résultats des tests présentés dans ces graphes montrent clairement que notre algorithme est plus rapide que ceux de la littérature et de J. Hempel. Ce dernier non seulement possède de très mauvais temps d'exécution dans le cas de notre problème mais aussi ne génère pas toutes les cliques.

Temps d'exécution en fonction de la densité



Temps d'exécution en fonction de la densité

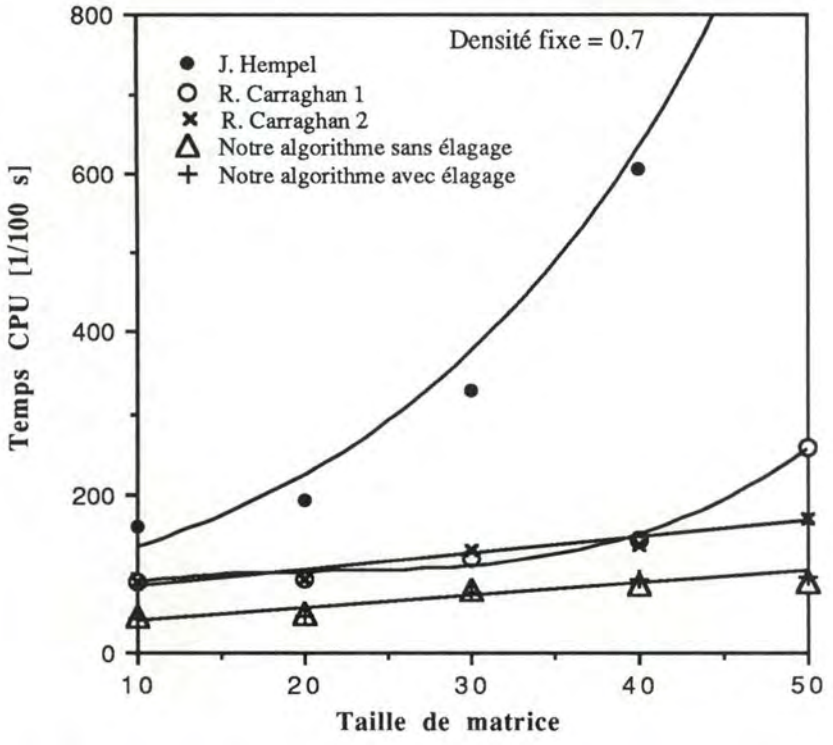


Certains tests montrent que les algorithmes de la littérature ne donnent pas systématiquement toutes les cliques non plus.

Etant donné l'exhaustivité de notre algorithme, dans le cas où il y a beaucoup de cliques, notre algorithme risque de prendre plus de temsp que les autres. ce phénomène n'est pas visible dans ces graphes.

Le dernier graphe montre le comportement des algorithmes en fonction de la taille de la matrice pour une densité constante de 0,7. Nous voyons clairement que "Brandnewcli" a un comportement exponentiel. C'est aussi le cas pour le programme de Carraghan où il n'y a pas d'ordonnancement. Le comportement de notre programme est relativement linéaire, en tout cas, pour les tailles de matrices que nous avons pu étudier.

Temps CPU en fonction de la taille de la matrice



On remarque également que notre critère d'arrêt n'a pas un effet très marqué pour ces petites matrices. Des études sur de grosses matrices seraient nécessaires pour mettre en évidence l'utilité de ce critère.

VII. Conclusion

A. Résultats obtenus

Notre algorithme semble être le plus rapide de tous les algorithmes que nous avons testés. De plus, c'est le seul qui, d'après le résultat de nos tests semble donner toutes les cliques maximums.

Le plus de notre algorithme réside dans le fait que nous exploitons les caractéristiques spécifiques du problème biologique étudié.

Le fait d'utiliser les groupes (recherche d'une clique maximum contenant un et un seul élément de chaque groupe) permet vraissemblablement d'accélérer la recherche de toutes les cliques maximums. Le nombre de sous-problèmes à étudier est réduit grâce à cette caractéristique.

A ce stade, nous pouvons tracer quelques évolutions futures qui devraient permettre d'améliorer encore les premiers résultats que nous avons obtenus.

B. Optimisations futures

La première optimisation à laquelle nous pouvons penser est d'utiliser une heuristique tenant compte du degré des sommets dans le graphe.

Deuxièmement, nous pourrions également donner des poids aux "matchs" sélectionnés qui se retrouvent dans la matrice. Nous pourrions alors avoir différents critères de sélection d'un candidat selon le degré de similarité. Ceci permettrait de faire tourner l'algorithme sur la matrice en étant plus ou moins exigeant sur la similarité.

Nous aurions une matrice remplie avec :

- 0 si les deux segments n'ont pas satisfait au critère de similarité;
- 1 si les deux segments ont fortement satisfait au critère de similarité;
- 11 si les deux segments ont satisfait moyennement au critère de similarité;
- 111 si les deux segments ont satisfait très moyennement au critère de similarité.

Dans le cas d'une grosse matrice, nous pourrions faire tourner l'algorithme avec des critères d'exigence progressifs sur la similarité.

Troisièmement, une étude plus approfondie devrait être réalisée sur l'importance des positions relatives des groupes au sein de la matrice. Il nous semble qu'il serait efficace de placer le groupe le plus grand en dernière position dans la matrice afin d'éviter un maximum d'appel récursifs.

Quatrièmement, la réalisation de tests préalables serait nécessaire pour éviter les solutions triviales (par exemple, dans la cas où un sommet est adjacent à tous les sommets des autres groupes ou encore où il n'est adjacent à aucun sommet).

Une dernière optimisation à laquelle nous avons pensé est l'utilisation du parallélisme.

VIII. Bibliographie

1. Aho A., Hopcraft J., Ulman J., "Structures de données et algorithmes", Inter Editions, Paris, 1987

2. Apostalico A., Atallah M.J.and Hambrusch S.E., "New Clique and Independent Set Algorithms for Circle Graphs", Discrete Applied Mathematics, 36 (1992), 1-24

3. Babel L., "Finding Maximum Cliques in Arbitrary and in Special Graph", Computing 46, 321-341(1991)

4. Balas E. and Yu C.S., "Finding a Maximum Clique in an Arbitrary Graph", SIAM J. COMPUT., vol.15, no.4, (november 1986) 1054-1068

5. Bränden C.I. & Tooze J. (1991), "Enzymes That Bind Nucleotides.", in "Introduction to Protein Structure.", Galand Publishing Inc., New York. London.

6. Bron C. and Kerbosch J., "Finding all Cliques of an Undirected Graph [H]", Communication of the A.C.M., sept.1973, vol.16, number 9, 575, 577.

7. Carraghan R., Pardalos P.M., "And exact Algorithm for the Maximum Clique Problem", "Operations Research Letters 9 (1990), 375-382

8. Christofides N., "Graph Theory an Algorithmic Approach", New York London, San Francisco, 1975, pp.32-35.

9. Clarke A.R., Colebrook S., Cortes A., Emery D.C., Halsall D.J., Hart K.W., Jackson R.M., Wilks H.M. & Holbrook J.J. (1991), "Towards the construction of a Universal $NAD(P)^+$-Dependent Dehydrogenase : Comparative and Evolutionary considerations.", Paper for Biochemichal Society Bioenergetics Group Colloquium on "Comparative Aspects of Respiratory Electron Transfer Complexes" April 10,1991 at Reading.

10. Cozzens M.B., Kelleher L.L., "Dominating Cliques in Graphs", Discrete Mathematics 86 (1990), 101-116

11. Depiereux E., Feytmans E., "Simultaneous and multivariate alignment of proteins sequences : correspondence between physicochemical profiles and structurally concerved regions (SCR)", Protein Engineering, vol. 4, n° 6, pp 603-613, 1991.

12. Erdos P., Gallai T.and Tuza Z., "Covering the Cliques of a Graph with Vertices", Discrete Mathematics 108 (1992), 279-289

13. Garey M.R. & Johnson D.S., "Computers and Intractability - A Guide to the Theory of NP-Completeness", Freeman, San Francisco, 1979

14. Gavril F., "Algorithms for a Maximum Clique and a Maximum Independent Set of a Circle Graph", Networks, 3 : 261-273 (1973)

15. Gerhards L. and Lindenberg W., "Clique Detection for Non directed Graphs : Two New Alborithms", Computing 21, 295-322 (1979)

16. Goldberg M. and Spencer T., "A New Parallel Algorithm for the Maximal Independent Set Problem", SIAM J. Comput., vol.18, no.2. pp.419-427, April 1989

17. Grindley H.M., Artymiuk P.Y., Rice D.W., Willet P., "Identification of Tertiary Structure Ressemblance in Proteins Using a Maximal Common Subgraph Isomaphism Algorithm", J. Mol.Biol. (1993), 229, 709-721

18. Grötschel M., Lovasz L., Schrijver A., "Geometric Algorithms and Combinatorial Optimization", Spinger-Verlag Berlin Herdelberg, 1988.

19. Gyarfas A., "A Simple Lower Bound on Edge Coverings by Cliques", Discrete Mathematics 85 (1990), 103-104

20. Hempel J.C. ,"A memo to Define Cliques and Applications of this concept in Conformatinal Analysis", Personnal communication (1990)

21. Joseph Y.- Leung T., "Fast Algorithms for Generating all Maximal Indepedent Sets of Interval, Circular-Arc and Chordal graphs"

22. Kashiwabara T.and Masuda S., Nakajima K.and Fujisawa T., "Generation of Maximum Independent Sets of a Bipartite Graph and Maximum Cliques of a Circular-Arc Graph", Journal of algorithms 13, 161-174 (1992)

23. Kratsch D., "Finding Dominating Cliques Efficiently, in Strongly Chordal Graphs and Undirected Path Graphs", Discrete Mathematics 86 (1990), 225-238

24. Kucera L., "Combinatorials Algorithms", (Faculty of Mathematics and Physics Charles University Czeckoslovakia ; Hilger A., Bristol and Philadelphia), 1990, p.11, 158-169, 180-185, 244-247.

25. Kulh F.S., Crippen G.M., Friesen DK., "A combinatorial Algorithm for Calculating Ligend Binding", Journal of Computational Chemistry, vol.5, no.1, 24-34 (1984)

26. Luby M., "A Simple Parallel Algorithm for the Maximal Independent Set Problem", SIAM J.COMPUT., vol.15, no.4, (november 1986) 1036-1053

27. Masuda S. and Nakajima K., "An Optimal Algorithm for Finding a Maximum Independent Set of a Circular-arc graph", SIAM J. COMPUT., vol.17, no.1, (february 1988) pp.41-52

28. Masuda S., Nakajima K., Kashiwabara T., Fujisawa T., "Efficient Algorithms for Finding Maximum Cliques of an Overlap graph", Networks, vol.20 (1990) 157-171

29. Melhorn K., "Data Structure and Algorithms 2 : Graph Algorithms and NP-Completeness.", Spinger-Verlag, Berlin Heidelberg-New York Tokyo, 1984, pp.171-173.

30. Murphy O.J., "Computing Independent Sets in Graphs with Large Girth", Discrete Applied Mathematics 35 (1992), 167-170

31. Reingold E.M., Nievergelt J., Narsingh D., "Combinatorials Algorithms : Theory and Practice", Prentice-Hall Inc. Englewoodcliefs New Jersey

32. Robson J.M., "Algorithms for Maximum Independent Set", Journal of Algorithms 7, 425-440 (1986)

33. Sedgewick R., "Algorithms in C.", Addison-Wesley, Publishing Company, Inc, 1990, pp.633-639., 415, 418-423.

34. Tarjan R.E.and Trojanowski A.E., "Finding a Maximum Independent Set", Siam J. comput, sept.1977, vol.6 , no.3, 537-546.

35. Tsukiyama S., Ide M., Ariyoshi H. and Shirakawa I., "A New Algorithm for Generating all the Maximal Independent sets", Siam J. comput, sept.1977, vol.6, no 3, 505-517.

36. Tuza Z., "Covering all Cliques of a Graph", Discrete Mathematics 86 (1990), 117-126

Index

Annexes

Annexe 1. Figures concernant la partie biologique

Les illustrations présentées dans cette annexe, sont tirées de "Biochemistry", Stryer L., printed in the United States of America, 1988.

Abbreviations for amino acids

Amino acid	*Three-letter abbreviation*	*One-letter symbol*
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic acid	Asp	D
Asparagine or aspartic acid	Asx	B
Cysteine	Cys	C
Glutamine	Gln	Q
Glutamic acid	Glu	E
Glutamine or glutamic acid	Glx	Z
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

Fig. II.1. Tableau des acides aminés avec leurs abréviations

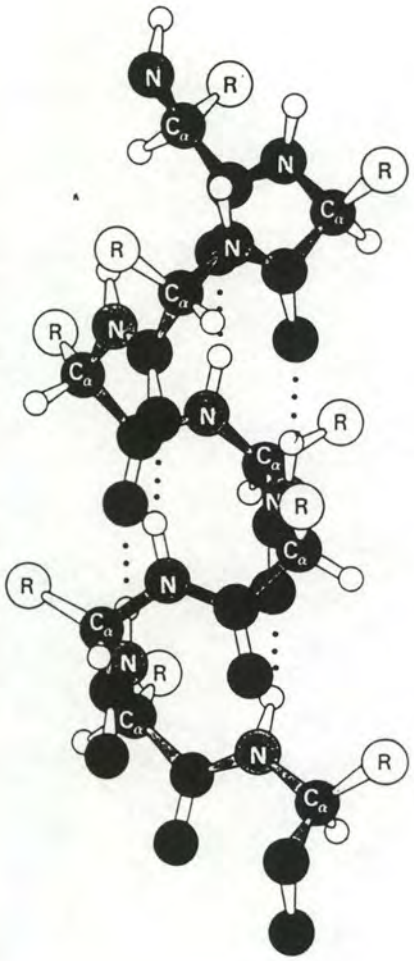


Fig. II.3. Hélice α

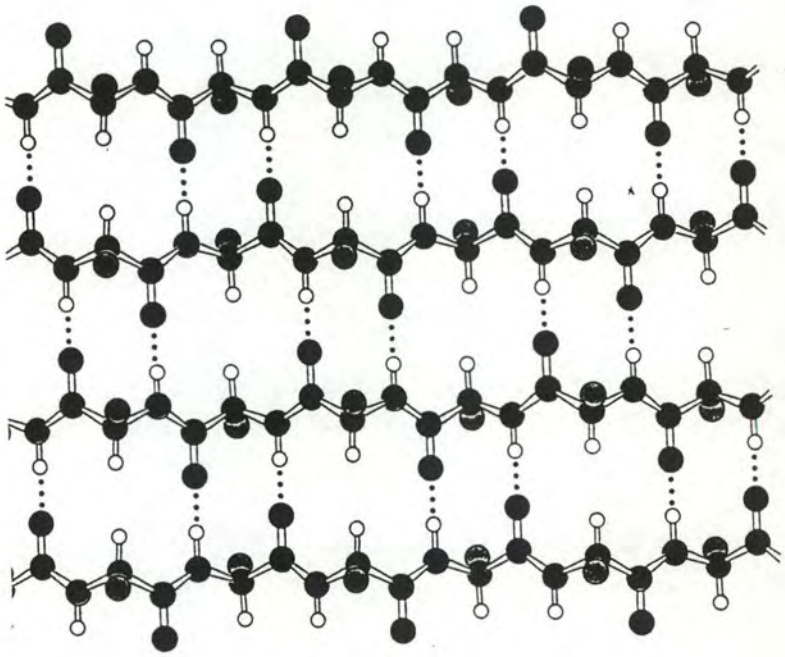


Fig. II.4. Feuillet β

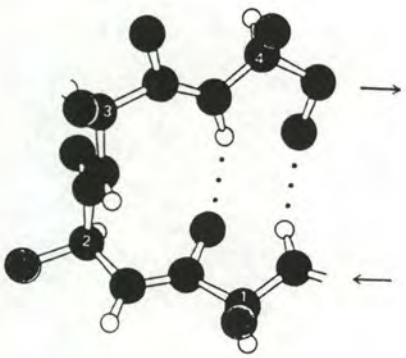


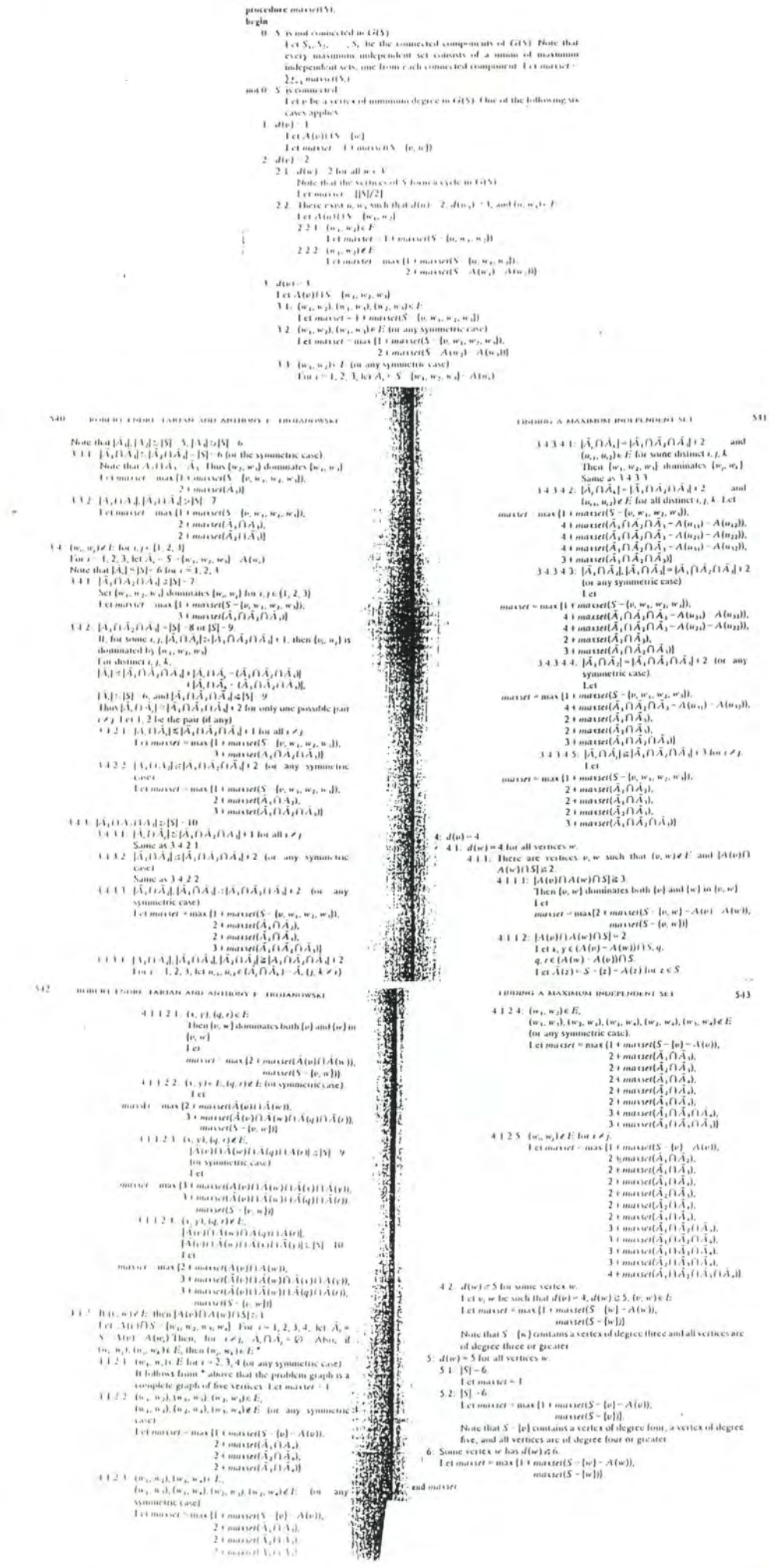
Fig. II.5. "turn"

Tableau résultat 2

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
1	T	G	cpu Br	cpul1	cpul2	cpuf1	cpuf2	nBr	nl1	nl2	nl2v	nf	Dens		H/C	I/D	J/E	K/F	L/G		S/O	S/P	S/Q	S/R
2	10	3	159	83	83	44	45	9	9	17	9	9	0,2		0,6	1,1	1,1	2,0	2,0		3,5	1,8	1,8	1,0
3	10	5	154	89	85	46	43	8	8	16	8	8	0,4		0,5	0,9	0,9	1,7	1,9		3,6	2,1	2,0	1,1
4	10	3	159	81	87	46	46	10	10	20	10	10	0,5		0,6	1,2	1,1	2,2	2,2		3,5	1,8	1,9	1,0
5	10	5	158	89	90	46	47	8	8	16	8	8	0,5		0,5	0,9	0,9	1,7	1,7		3,4	1,9	1,9	1,0
6	10	3	159	89	82	50	43	18	18	34	18	18	0,6		1,1	2,0	2,2	3,6	4,2		3,7	2,1	1,9	1,2
7	10	5	160	81	92	45	46	9	9	18	9	9	0,6		0,6	1,1	1,0	2,0	2,0		3,5	1,8	2,0	1,0
8	10	3	159	91	91	46	43	12	12	24	12	12	0,7		0,8	1,3	1,3	2,6	2,8		3,7	2,1	2,1	1,1
9	10	5	155	88	88	46	51	15	16	32	16	16	0,7		1,0	1,8	1,8	3,5	3,1		3,2	1,7	1,7	0,9
10	20	2	166	92	90	48	41	26	26	26	26	26	0,2		1,6	2,8	2,9	5,4	6,3		4,0	2,2	2,2	1,2
11	20	2	174	90	91	52	54	57	57	57	57	57	0,3		3,3	6,3	6,3	11,0	10,6		3,2	1,7	1,7	1,0
12	20	5	173	90	86	47	48	0	0	0	0	0	0,3		0,0	0,0	0,0	0,0	0,0					
13	20	2	180	93	97	49	53	70	70	89	70	70	0,4		3,9	7,5	7,2	14,3	13,2		3,4	1,8	1,8	0,9
14	20	7	171	92	88	51	49	0	0	0	0	0	0,4		0,0	0,0	0,0	0,0	0,0					
15	20	5	176	91	91	46	44	10	10	20	10	10	0,5		0,6	1,1	1,1	2,2	2,3		4,0	2,1	2,1	1,0
16	20	7	187	92	97	49	53	9	10	20	10	10	0,5		0,5	1,1	1,0	2,0	1,9		3,9	1,7	1,8	0,9
17	20	5	187	96	92	51	52	17	18	36	18	18	0,6		0,9	1,9	2,0	3,5	3,5		3,8	1,8	1,8	1,0
18	20	7	183	93	95	47	52	8	8	15	8	8	0,6		0,4	0,9	0,8	1,7	1,5		3,5	1,8	1,8	0,9
19	20	5	205	96	97	60	57	46	51	102	51	51	0,7		2,2	5,3	5,3	8,5	8,9		4,0	1,7	1,7	1,1
20	20	7	192	94	93	50	47	3	3	6	3	3	0,7		0,2	0,3	0,3	0,6	0,6		4,1	2,0	2,0	1,1
21	30	2	186	98	102	53	56	42	42	42	42	42	0,1		2,3	4,3	4,1	7,9	7,5		3,3	1,8	1,8	0,9
22	30	3	184	99	98	52	54	13	13	13	13	13	0,2		0,7	1,3	1,3	2,5	2,4		3,4	1,8	1,8	1,0
23	30	5	139	95	94	52	57	6	6	6	6	6	0,2		0,4	0,6	0,6	1,2	1,1		2,4	1,7	1,6	0,9
24	30	2	207	107	108	61	59	138	133	138	138	138	0,3		6,7	12,4	12,8	22,6	23,4		3,5	1,9	1,8	1,0
25	30	2	210	111	104	62	67	112	110	112	112	112	0,3		5,3	9,9	10,8	18,1	16,7		3,1	1,7	1,6	0,9
26	30	7	206	99	94	51	49	0	0	0	0	0	0,3		0,0	0,0	0,0	0,0	0,0					
27	30	10	210	96	102	51	47	0	0	0	0	0	0,3		0,0	0,0	0,0	0,0	0,0					
28	30	2	226	103	108	65	65	160	149	160	160	160	0,4		7,1	14,5	14,8	24,6	24,6		3,5	1,7	1,7	1,0
29	30	3	230	106	103	79	70	159	158	159	159	159	0,4		6,9	14,9	15,4	20,1	22,7		3,3	1,5	1,5	1,1
30	30	3	255	110	123	81	86	235	229	377	235	235	0,5		9,2	20,8	19,1	29,0	27,3		3,0	1,3	1,4	0,9

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
31	30	3	308	118	128	102	100	350	310	398	350	350	0,5		11,4	26,3	27,3	34,3	35,0		**3,1**	**1,3**	**1,3**	**1,0**
32	30	5	228	94	104	55	59	28	31	62	31	31	0,5		1,2	3,3	3,0	5,6	5,3		**4,3**	**1,6**	**1,8**	**0,9**
33	30	7	229	94	96	51	46	0	0	0	0	0	0,5		0,0	0,0	0,0	0,0	0,0					
34	30	10	238	98	101	48	54	0	0	0	0	0	0,5		0,0	0,0	0,0	0,0	0,0					
35	30	5	255	108	125	82	79	92	100	254	127	127	0,6		3,6	9,3	10,2	15,5	16,1		**4,5**	**1,7**	**1,6**	**1,0**
36	30	7	268	106	101	52	59	8	11	22	11	11	0,6		0,3	1,0	1,1	2,1	1,9		**6,2**	**1,8**	**1,7**	**0,9**
37	30	7	238	93	100	48	47	0	0	0	0	0	0,6		0,0	0,0	0,0	0,0	0,0					
38	30	10	270	99	96	51	53	0	0	0	0	0	0,6		0,0	0,0	0,0	0,0	0,0					
39	30	5	339	136	155	194	183	289	411	596	596	634	0,7		8,5	30,2	38,5	32,7	34,6		**4,1**	**1,1**	**0,9**	**1,1**
40	30	7	330	119	131	80	76	58	90	180	90	90	0,7		1,8	7,6	6,9	11,3	11,8		**6,7**	**1,6**	**1,7**	**1,1**
41	30	10	300	103	99	53	54	5	6	12	6	6	0,7		0,2	0,6	0,6	1,1	1,1		**6,7**	**1,9**	**1,8**	**1,0**
42	40	8	237	106	104	49	51	0	0	0	0	0	0,2		0,0	0,0	0,0	0,0	0,0					
43	40	8	357	105	106	56	54	0	0	0	0	0	0,5		0,0	0,0	0,0	0,0	0,0					
44	40	8	457	118	113	58	55	1	1	2	2	1	0,6		0,0	0,1	0,2	0,2	0,2		**8,3**	**2,1**	**1,0**	**1,1**
45	40	8	850	192	214	418	411	262	356	639	639	979	0,7		3,1	18,5	29,9	23,4	23,8		**7,7**	**1,3**	**0,8**	**1,0**
46	40	8	608	142	135	86	93	35	67	134	67	67	0,7		0,6	4,7	5,0	7,8	7,2		**12,5**	**1,5**	**1,5**	**0,9**
47	50	10	306	116	117	56	59	0	0	0	0	0	0,2		0,0	0,0	0,0	0,0	0,0					
48	50	15	380	134	132	61	55	0	0	0	0	0	0,3		0,0	0,0	0,0	0,0	0,0					
49	50	25	2528	728	185	56	59	0	0	0	0	0	0,5		0,0	0,0	0,0	0,0	0,0					
50	50	10	595	129	125	56	58	0	0	0	0	0	0,5		0,0	0,0	0,0	0,0	0,0					
51	50	15	720	163	132	54	59	0	0	0	0	0	0,5		0,0	0,0	0,0	0,0	0,0					
52	50	10	804	147	129	58	62	0	0	0	0	0	0,6		0,0	0,0	0,0	0,0	0,0					
53	50	15	1018	117	135	58	58	0	0	0	0	0	0,6		0,0	0,0	0,0	0,0	0,0					
54	50	25	2758	404	157	57	61	0	0	0	0	0	0,6		0,0	0,0	0,0	0,0	0,0			-		
55	50	10	1203	258	170	89	96	7	53	53	53	53	0,7		0,1	2,1	3,1	6,0	5,5		**94,9**	**2,7**	**1,8**	**0,9**
56	50	25	2735	1179	200	55	60	0	0	0	0	0	0,7		0,0	0,0	0,0	0,0	0,0					
57	50	10	2004	510	522	511	345	179	464	598	598	916	0,8		0,9	9,1	11,5	17,9	26,6		**29,7**	**2,9**	**2,3**	**1,5**
58																								
59																								
60																					*7,5*	*1,8*	*1,7*	*1,0*

Annexe 3. Pseudo-code de la littérature



Annexe 4. Programmes

rancli.for

```
      program ransym

c     genereate a random truth matrix

      integer dim,cli
      real x(1000,1000)
      open (unit=40,file='truth.dat',access='sequential',status='old')

      open (unit=42,file='seed.dat',access='sequential',status='old')
      open (unit=43,file='donnee.dat',access='sequential',status='new')

      write (6,*) ' enter the dimension of the truth matrix and return...'
      read (6,*) dim
      write (6,1002) dim
1002        format (' enter a number 1<n<',I4,
     1      ' to force a given dimension of clique')
      read (6,*) cli
      write (6,*) ' enter a delta from .1 to .9 to fix the ratio 0/1'
      read (6,*) delta
      write(43,1008) dim, cli, delta
1008        format (' taille de la matrice:',I4,', de la clique max.:',I4,
     1      ', ratio 0/1:',F4.2)

c     need a number to initialize the randomization. If seed.dat does not
c     exist, create it with a single great odd integer

      read (42,*) seed
      rewind(42)

      x(dim,dim)=1
      do 10 i=1,dim
      do 10 j=i+1,dim
      x(i,i)=1

c     this branch to force a given size of clique
      if ((i.le.cli).and.(j.le.cli)) then
      x(i,j)=1
      go to 8
      endif
      x(i,j)=int(ranvax(seed)+delta)
8     x(j,i)=x(i,j)
10    continue

c     save the last value of seed for the next run
      write (42,*) seed

      write (40,*) dim

      do 20 i=1,dim
20    write(40,1000) (x(i,j),j=1,dim)

1000  format (2x,<DIM>F2.0)
```

```
1001    format (2x,<DIM>i2)
      close(43)
      end

c     function ranvax obtained from the source of discover

      function ranvax(seed)
      implicit double precision (a-h,o-z)
      data rmagic/69069.D0/,twop32/42949667296.d0/,one/1.0d0/
      seed=mod(rmagic*abs(seed)+one,twop32)
      ranvax=seed/twop32
      return
      end
```

mixcli.for

```
      program mixcli

c     sort a matrix at random and conserve reference of the original
c     ordering

      integer res(300,2),dim
      real truth(300,300)
      open (unit=40,file='truth.dat',access='sequential',status='old')
      open (unit=41,file='truth1.dat',access='sequential',status='new')
      open (unit=42,file='seed.dat',access='sequential',status='old')
c
c     read a number to initialize the randomization. If seed.dat
c     does not exist, create it and write in it a single great odd
c     integer

      read(42,*)  seed
      rewind(42)

c     read the parameters and the truth matrix

      read (40,*) dim


      do 5 i=1,dim
5     read(40,1001) (truth(i,j),j=1,dim)

1000  format (2x,<dim>i2)
1001    format (2x,<dim>f2.0)

c     generate random values (integer)

      do 10 i=1,dim
      res(i,1)=int(ranvax(seed)*100)
10    res(i,2)=i

c     save the last value of seed for the next run of the program
      write(42,*) seed

c     sort the results


      CHANGE=dim-1
115   REMEMBER=1
      DO 120 I=1,CHANGE
      IF (RES(I,1).LE.RES(I+1,1)) GO TO 120

            DO 130 K=1,2
            TEMP1=RES(I,K)
            RES(I,K)=RES(I+1,K)
            RES(I+1,K)=TEMP1
            REMEMBER=I
130         CONTINUE
120         CONTINUE
      CHANGE=REMEMBER

      IF (REMEMBER.GT.1) GO TO 115
```

```
c     write results

      write(41,*) dim
      write(41,1000) (res(i,2),i=1,dim)
      do 500, i=1,dim
500   write(41,1001) (truth(res(i,2),res(j,2)),j=1,dim)
      end

c     function ranvax obtained from the source of Discover

      function ranvax(seed)
      implicit double precision (a-h,o-z)
      data rmagic/69069.D0/,twop32/42949667296.d0/,one/1.0d0/
      seed=mod(rmagic*abs(seed)+one,twop32)
      ranvax=seed/twop32
      return
      end
```

Convers.for

```
      program makemat
      integer indice(99)
      integer nseq
      integer nmatch(99)
      real mat(99,99)
      open (unit=40,file='truth1.dat',access='sequential',status='old')
      open (unit=41,file='mat.dat',access='sequential',status='old')
      open (unit=42,file='indice.dat',access='sequential',status='old')
      open (unit=43, file='donnee2.dat',access='sequential',status='new')
C-------------------------------------------------------------------------
-
C     MATRICE NON MODIFIEE (COPIE DE TRUTH1.DAT)
C_________________________________________________________________________
_
      read(40,*) NCONF
      read (40,432) (indice(j),j=1,nconf)
432   format (2x,<nconf>i2)
4320  format (i2)
4321  format (2x,i2)
      do 5 i=1,nconf
      read (40,530) (mat(i,j),j=1,nconf)
5       continue
530   format (2x,<nconf> f2.0)
C-------------------------------------------------------------------------
----
C     PARTIE POUR DEMANDER LE NOMBRE DE MATCH PAR SEQUENCE
C_________________________________________________________________________
____

      matchs = nconf
      write (6,1003) (matchs)
      write (6,*) ' entrer le nombre de sequences puis RETURN'
      read (6,*) nseq
      write (43,*) nseq
1600  format  (' nombre de sequences: ',I2)
      nmatch(1) = 0
        write (42,4321) nseq
      matchs = nconf
      do 10 k=2,nseq+1
         write (6,1003) (matchs)
1003  format  (' nombre de matchs restants: ',I2)
         write (6,1000) (k-1)
1000  format  (' entrer le nombre de matchs dans la sequence',I2)
         read (6,*) ih
         matchs = matchs - ih
         write (43,1620) (k-1), ih
1620  format  (' nombre de matchs dans la sequence ',I2, ' = ', I2)
         nmatch(k) = nmatch(k-1) + ih
           write (42, 4321) ih
10    continue
        close(42)
      close(43)
C-------------------------------------------------------------------------
-----
C     PARTIE POUR MODIFIER LA MATRICE
C
```

```
C-------------------------------------------------------------------------
-----
      x=0
      do 15 i=1,nseq
            do 20 j=(nmatch(i)+1), nmatch(i+1)
                do 30 l=(nmatch(i)+1), nmatch(i+1)
                  if (j.eq.l) mat(j,l) = 1
                  if (j.ne.l) mat(j,l) = 0
30              continue
20          continue
15    continue
C-------------------------------------------------------------------------
-----
C     PARTIE POUR RECOPIER LA MATRICE MODIFIEE
C
C-------------------------------------------------------------------------
-----

      write (41,*) nconf
      write(41,1001) (indice(j),j=1,nconf)
C
1001    format (2x,<nconf>i2)
C
      do 40 i=1, nconf
           write(41,2000) (mat(i,j),j=1,nconf)
2000  format (2x,<nconf> f2.0)
40    continue
      end
```

Densite.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define true 1
#define false 0

int *mat, *column, *density, N;

int density_matrix(void);
void reset_matrix(void);


int density_matrix(void)
{
  int i, j;
  char matrixname[25], buf[11];
  FILE *data, *fopen();

/*  printf("Entrer le nom du fichier contenant la matrice -> ");*/
/*  scanf("%s", matrixname);*/
  strcpy(matrixname, "mat.dat");
  data = fopen(matrixname, "r");
  if (data != NULL) {
    for (i = 0; i < N; i++)
      density[i] = 0;
    fread(buf, 10, 1, data);      /* 10 blancs */
    fread(buf, 2, 1, data);       /* Taille de la matrice */
    buf[2]='\0';
    N = atoi(buf);
    fread(buf, 3, 1, data);       /* 2 blancs + 0D + 0A */
    density = (int *) malloc (sizeof(int) * N);
    for (i = 0; i < N; i++)
      density[i] = 0;
    column = (int *) malloc (sizeof(int) * N);
    for (i = 0 ; i < N ; i++) {
      fread(buf, 2, 1, data);
      buf[2]='\0';
      column[i] = atoi(buf);      /* Indices des colonnes */
    }
    mat = (int *) malloc (sizeof(int) * N * N);
    reset_matrix();
    for (i = 0 ; i < N ; i++) {
      fread(buf, 3, 1, data);      /* 2 blancs + 0D + 0A */
      for (j = 0 ; j < N ; j++) {
      fread(buf, 2, 1, data);   /* Valeurs des eÇlÇements de la matrice */
      buf[1] = '\0';
      mat[i * N + j] = atoi(buf);
      density[i] = density[i] + mat[i * N + j];
      }
    }
    fclose(data);
    return 0;
  }
  else {
    return 1;
  }
}
```

```
void reset_matrix()
{
  int i, j;

  for (i = 0 ; i < N ; i++) {
    for (j = 0 ; j < N ; j++) {
      mat[i*N+j] = 0;
    }
  }
}

main()
{
  int i;
  float sumdens;
  sumdens = 0;
  if (!density_matrix()){
      for (i = 0; i < N; i++)
      sumdens = sumdens + density[i];
      printf("La densite de la matrice est : %f", sumdens/(N*N));
  }
}
```

Brandnewcli.for

```
C     PROGRAM CLIQUE +++++ CLICKER ++++++ ERIC'S ADDITIONS = ECLICKER

c              third version - may 90
C
C--THIS PROGRAM IS INTENDED TO BE JUST A DRIVER PROGRAM FOR CALLING
C--THE SUBROUTINES THAT I HAVE DEVELOPED TO IDENTIFY ALL CLOSED SETS
C--OF CONFORMERS (FOR EXAMPLE) WERE EACH CONFORMER IS RELATED TO EVERY
C--OTHER MEMBER OF THE SET IN THE SAME WAY
C
C--FOR EXAMPLE--
C--I.   EVERY CONFORMER A MEMBER OF THE SET IS SIMILAR TO WITHIN
C--   A LEAST SQUARES RMS DEVIATION OF XX A TO EVERY OTHER (RMS DEFINED
C--   OVER THE SAME SET OF ATOMS)
C
C--II.  EVERY TEMPLATE (A MEMBER OF A CLOSED SET) IS LINKED BY NOES TO
C--   EVERY OTHER MEMBER OF THE SET BY ONE OR MORE NOES
C
C--OR,  IN A GENERAL SENSE EVERY MEMBER OF THE SET IS RELATED TO EVERY
C--   OTHER BY A RULE (VERY MUCH LIKE A GROUP OPERATION THAT IS USED TO
DEFINE
C--   A GROUP AND IN FACT THIS MAY ACTUALLY BE SO AND SOMEDAY I INTEND TO
C--   DEFINE THIS GROUP CONCEPT FURTHER)
C
C--THE ALGORITHM USES A "TRUTH TABLE" OF THE FOLLOWING FORM WHERE THE TABLE
C--IS A SQUARE MATRIX WITH ROWS AND COLUMNS CORRESPONDING TO AN INDEX THAT
C--IDENTIFIES THE THINGS UNDER ANALYSIS (E.G. THE CONFORMERS IN EXAMPLE ONE
C--ABOVE, THE TEMPLATES IN EXAMPLE TWO, ETC...)
C
C     1     2     3     ...   N
C  1  1     1     1  ... 0  ... 1
C  2  1     1     0     0     1
C  3  1     0     1     0     1
C ...
C  N  1     1     1     0     1
C
C--IN THIS EXAMPLE 2 IS NOT "LIKE" 3 AND TWO CLOSED SETS ARE ILLUSTRATED:
C
C     1,2,N
C AND
C     1,3,N
C
C--AS  THIS  EXAMPLE  ILLUSTRATES  THE  CLOSED  SETS  (=CLIQUES)  ARE  NOT
NECESSARILY
C--DISJOINT (A THING CAN BELONG TO MORE THAN ONE CLIQUE)
C
C--THE ALGORITHM USED IN THIS PROGRAM IDENTIFIES ALL SUCH CLIQUES GIVEN A
C--TRUTH TABLE DEFINED FOR THE THINGS UNDER ANALYSIS SUBJECT TO THE RULE OF
C--INTEREST FOR THE ANALYSIS OF THESE THINGS
C
c
c--at this time it is dimensioned so that
c--we can analyze 99 THINGS
c
      common/d/isub(99,99),idmat(99,99)
      COMMON/DD/dmetric(50,50,99),iconf,nconf,nres
      common/a/ila(8000),ilb(8000),dist(99,99)
      COMMON/BB/DB(99,99)
```

```
       common/bbb/dstart(99,99)
       COMMON/BX/NTEMP
       common/cc/teng(99)
       common/zz/indice(99)
      COMMON/CKK/ICLICKS(10000,99),NCLICKS
      common/ccc/ilister(10000,99),irowlist
C
c--THE TESTS WILL BE ADMINISTERED TO A MATRIX WITHIN THIS
C--SET OF SUBROUTINES
C--HENCE THE FIRST STEP IS TO FILL
C--TWO MATRICES, THE DISTANCE (REFERNCE) MATRIX AND THE
C--DSTART ("FIRST STEP") MATRIX WITH THE ENTRIES IN
C--WHICH PASS THE TEST. IN THIS EXAMPLE
C--THE MATRIX TO BE ANALYZED IS A SQUARE MATRIX.
C
      OPEN (UNIT=4,FILE='judy.DAT',ACCESS='SEQUENTIAL',STATUS='old')
      open (unit=41,file='eric.dat',access='sequential',status='old')
      open (unit=42,file='indice.dat',access='sequential',status='old')
c     OPEN (UNIT=6,FILE='judy.clique',ACCESS='SEQUENTIAL',STATUS='OLD')
      OPEN (UNIT=40,FILE='mat.DAT',ACCESS='SEQUENTIAL',STATUS='OLD')

      read (42,*) Ngroupe
      Ncliques = 0
      IRES=1
      KRES=1
      read(40,*) NCONF
        IW=6
        idsum=0

      read (40,432) (indice(j),j=1,nconf)

        do 40 i=1,Nconf
        do 40 j=1,Nconf
      idmat(i,j)=0
        DIST(I,J)=0.0
        DSTART(I,J)=0.0
   40   continue


      DO 43, I=1,NCONF
      READ (40,431) (DIST(I,J),J=1,NCONF)
c     WRITE (5,431) (DIST(I,J),J=1,NCONF)
   43   CONTINUE
      do 44, i=1,nconf
      do 44, j=1,nconf
44    dstart(i,j)=dist(i,j)
c     do 45 i=1,nconf
c     write (5,431) (dstart(i,j),j=1,nconf)
c   45   continue
  431   FORMAT (2x,<NCONF>F2.0)
 432  format (2x,<nconf>i2)
c
c--i am adding this print for eric so that he can check the algorithm
c--for me
ccccccccccccccc
c
 134  format(10i5)
c     do 132 ig=1,nconf
c     write(6,131)(ig,(dstart(ig,jg),jg=1,nconf))
c 132 continue
c     write(6,133)
 133  format(/)
 131  format(i3,2x,60f2.0)
```

```
c
c--begin to check for sets of related distances
C
C
      irowlist=0
      NCLICKS=0
        IFAM=0
        do 220 irow=1,nconf
      itsachk=0
c
c     do 632 ig=1,nconf
c     write(6,131)(ig,(dist(ig,jg),jg=1,nconf))
c 632 continue
C
C--CHECK TO SEE IF THERE ARE NO ENTRIES LINKING THIS CONFORMER TO ANOTHER
C
          DO 210 KCOL=1,nconf
          IF (IROW .EQ. KCOL) GO TO 210
          IF (DIST(IROW,KCOL) .NE. 0) GO TO 217
  210     CONTINUE
          ICNT=1
          ILA(1)=IROW
           IP=2
           IFAM=IFAM+1
      irtest=0
      ires=irow
      kres=kcol
c     write(6,559)ires,kres
      CALL SETWRITE(nconf,ISET,TEST,ICNT,IP,IFAM,ires,kres,
     1  irtest,ngroupe,ncliques)

      go to 220
  217     CONTINUE
       new=irow+1
        do 200 icol=new,nconf
c
cc    do 812 iqq=icol,nconf
cc    do 812 jqq=icol,nconf
cc    dstart(iqq,jqq)=dist(iqq,jqq)
      do 812 jqq=1,nconf
      dstart(irow,jqq)=dist(irow,jqq)
 812  continue
      do 813 iqq=1,nconf
      dstart(iqq,icol)=dist(iqq,icol)
 813  continue
C
C--CHECK TO SEE IF CLICKS HAVE BEEN DEFINED PREVIOUSLY INCORPORATING THIS
C--STARTING POINT
C
      DO 101 ICKCH=1,NCLICKS
c     write(6,559)(iclicks(ickch,iug),iug=1,nconf)
c     write(6,559)(ilister(ickch,iug),iug=1,nconf)
      IF (ICLICKS(ICKCH,IROW) .EQ. 0) GO TO 101
      IF (ICLICKS(ICKCH,ICOL) .EQ. 0) GO TO 101
      NUMNUM=0
c     write(6,559)irow,icol
      DO 202 JCHK=1,nconf
      IF (Iclicks(ICKCH,JCHK) .EQ. 1) numnum=numnum+1
 202  CONTINUE
 203  CONTINUE
      DO 303 JCHK=1,NUMNUM
      JJJJ=ILISTER(ICKCH,JCHK)
      DO 303 KCHK=1,NUMNUM
```

```
      KKKK=ILISTER(ICKCH,KCHK)
      DSTART(JJJJ,KKKK)=0
 303  CONTINUE
 101  CONTINUE
 505  CONTINUE
C
        CALL RESET(nconf)
        icnt=0
c
CCNEWTEST
CCNEWTEST           if ( dstart(irOW,icOL) .gt. 0.0) then
          if ( dIST(irOW,icOL) .gt. 0.0) then
c     IF (IROW .GE. 17) THEN
c     WRITE(4,559)IROW,ICOL
c     WRITE(4,5779)(DSTART(IROW,IWRT),IWRT=1,NCONF)
 5779 FORMAT(10F10.1)
c     ENDIF
C
  772           ila(1)=irow
               ila(2)=icol
               icnt=2
      itsachk=itsachk+1
c     write(6,559)itsachk
c
               call threeset(nconf,irow,icol,icnt)
c     IF (IROW .GE. 17) THEN
c     write(6,559)(irow,icol,icnt,(ila(jjk),jjk=1,icnt))
c     ENDIF
      ifirstcnt=icnt
 559  format(20i5)
CC
CC--FOR THE CLUSTER ANALYSIS WE WANT TO KEEP ENTRIES ARISING FROM
CC--A SINGLE DIAGONAL ENTRY IN THE NOE TEMPLATE INTERSECTION MATRIX
CC
             IF (ICNT .eq. 2) THEN
      IF (dstart(irow,icol) .eq. 0) GO TO 200
      dstart(irow,icol)=0
      dstart(icol,irow)=0
      GO TO 206
           ENDIF
  201   FORMAT(10I5)
           call MOREset(nconf,irow,icol,icnt)
c     write(6,559)icnt
c     write(6,559)(ila(iuh),iuh=1,icnt)
  206      IP=2
           IFAM=IFAM+1
      irtest=irow
      ires=irow
      kres=icol
      CALL SETWRITE(nconf,ISET,TEST,ICNT,IP,IFAM,ires,kres,
     1  irtest,ngroupe,ncliques)
c
cc    do 132 ig=1,nconf
cc    write(6,131)(ig,(dstart(ig,jg),jg=1,nconf))
cc 132     continue
cc    write(6,133)
c
      go to 772
           ENDIF
  200   continue
  220 continue
      write(41,271) Ncliques
  271   FORMAT(' Nombre de cliques pour Brandnewcli',I5)
```

```
          end
c
c
          subroutine writesublock(kset,ione,none,itwo,ntwo,iw)
c
c--this subroutine writes out the  analysis matrix
c--by block
       common/d/isub(99,99),idmat(99,99)
       COMMON/DD/dmetric(50,50,99),iconf,nconf,nres
       dimension ilab(99),ILC(99)
c
          if (iw .eq. 9) then
          write(iw,691)
  691     FORMAT(1H1)
C
          icnt=0
  599     FORMAT(/)
          do 696 i=ione,none
          ICNT=ICNT+1
          ilab(iCNT)=i
  696     continue
          write(iw,692)(ilab(i),i=1,icnt)
  692     format(/' ROWS:',40I3)
          ICNT=0
          DO 697 I=ITWO,NTWO
          ICNT=ICNT+1
          ILAB(ICNT)=I
  697     CONTINUE
          write(iw,693)(ilab(i),i=1,icnt)
  693     format(' COLS:',40I3)
          WRITE(iw,599)
c
          endif
c
          do 699 i=ione,none
          WRITE(iw,904)(I,(IDMAT(I,J),J=itwo,ntwo))
  904     format(I4,':',58i2/5X,58I2/5X,58I2)
           ISUM=0
  699     CONTINUE
  900     continue

          end
c
        subroutine threeset(nATOM,irow,icol,icnt)
CX
cxNEWCLIQUE 2/28/90
CX
       common/a/ila(8000),ilb(8000),dist(99,99)
       COMMON/DD/dmetric(50,50,99),iconf,nconf,nres
       COMMON/BB/DB(99,99)
       common/bbb/dstart(99,99)
       COMMON/BX/NTEMP
       common/cc/teng(99)
      COMMON/CKK/ICLICKS(10000,99),NCLICKS
      common/ccc/ilister(10000,99),irowlist
c
c--the irow and icol identifiers are defined by the
c--dstart matrix
c--the dstart matrix = matrix with preset distances
c--from which we eliminate matrix elements within
c--previously defined "block diagonal" matrix
c--sets as starting points for searches for further
c--"block diagonal" matrix distance sets
```

```
c
c--the dist matrix = matrix with "known" distances and
c--zero entries for "unknown" distances
c
c--THE ALGORITHM:
c--given a non-zero "known" distance matrix entry
c--check to see if there are "known" distance matrix
c--entries connecting the two
c
        IW=8
        JJ=1
CCVV  jj=irow
        do 100 ii = jj,NATOM
c
        IF (IROW .EQ. II) GO TO 100
      if (icol .eq. ii) go to 100
c
c--checking each column of row corresponding to the
c--column position of initial non-zero entry
c--to see if there is a non-zero entry there
c
CCNEWTRY
CCV
c           if ( dstart(irow,icol) .eq. 0.0) then
        if ( dSTART(ii,icol) .gt. 0.0001) then
c
c--then....check to see if there is an entry linking
c--the new and the original non-zero entries
c
        IF (ICOL .EQ. II) GO TO 100
        if ( dist(irow,ii) .gt. 0.0001) then
        icnt=icnt+1
        ila(icnt)=ii
c     ix=1
c     write(6,333)irow,icol,ii,ix
c 333 format(10i5)
        return
        endif
        endif
c           endif
c           if (dstart(irow,icol) .gt. 0.001) then
CCXXNEX
c         IF (ICOL .EQ. II) GO TO 100
c     IF (II .LT. ICOL) DEST=DSTART(II,ICOL)
c     IF (II .GT. ICOL) DEST=DIST(II,COL)
CCXXNEX         if ( dist(ii,icol) .gt. 0.0001) then
c         if ( dEST .gt. 0.0001) then
c
c--then....check to see if there is an entry linking
c--the new and the original non-zero entries
c
ccnewtry
c         if ( dist(irow,ii) .gt. 0.0001) then
c         icnt=icnt+1
c         ila(icnt)=ii
c     ix=2
c     write(6,333)irow,icol,ii,ix
c       return
c        endif
c        endif
c           endif
  100  continue
       end
```

```
c
       subroutine MOREset(NATOM,irow,icol,icnt)
CX
cxNEWCLIQUE 2/28/90
CX
       common/a/ila(8000),ilb(8000),dist(99,99)
       COMMON/DD/dmetric(50,50,99),iconf,nconf,nres
       COMMON/BB/DB(99,99)
       common/bbb/dstart(99,99)
       COMMON/BX/NTEMP
       common/cc/teng(99)
      COMMON/CKK/ICLICKS(10000,99),NCLICKS
      common/ccc/ilister(10000,99),irowlist
c
c
c--here we start with the knowlege that there are
c--nonzero entries for dist(irow,icol),dist(irow,IX)
c--and dist(icol,IX) WHERE IX=ILa(ICNT)
c
c--the goal is to first check to see if there are additional
c--sets of non zero distance within column
c
        IW=8
        jj=1
        do 100 ii = jj,NATOM
        ilimit=icnt
c
c--the first time through the loop icnt=3
c
c--here we check for nonzero connections with irow
c
       if ( dist(irow,ii) .gt. 0.0) then
c
c--then to check to see if all rows identified so far have
c--nonzero linking values to ii
c
          DO 50 IR=1,Ilimit
          Irr=ila(ir)
          if ( dist(irr,ii) .EQ. 0.0) then
          GO TO 100
          ENDIF
  50     continue
          icnt=icnt+1
          ila(icnt)=ii
       endif
  100  continue
        end
C
C
      SUBROUTINE SETWRITE(NATOM,ISET,TEST,ICNT,IP,IFAM,ires,kres,
     1  irtest,ngroupe,ncliques)
CX
cxNEWCLIQUE 2/28/90
CX
       DIMENSION ILC(99),AVGMET(99)
       common/d/isub(99,99),idmat(99,99)
       common/a/ila(8000),ilb(8000),dist(99,99)
       COMMON/BB/DB(99,99)
       common/bbb/dstart(99,99)
       COMMON/DD/dmetric(50,50,99),iconf,nconf,nres
       COMMON/BX/NTEMP
       common/cc/teng(99)
      COMMON/CKK/ICLICKS(10000,99),NCLICKS
```

```
      common/ccc/ilister(10000,99),irowlist
      common/zz/indice(99)
c
c
c--set the "counting" matrix to zero
c
        IW=6
C
         IF (ICNT .EQ. 1) THEN
         ILC(1)=ILA(1)
         JCNT=1
         GO TO 333
         ENDIF
C
ccv
ccv         DO 40 J=1,natom
         DO 40 K=1,natom
ccv         DB(J,K)=0.0
      db(1,k)=0
   40    CONTINUE
c
c--fill the counting matrix with hits from
c--path search
c
          DO 50 I=1,ICNT
          IR=ILA(I)
ccv          imax=i+1
ccv          do 50 ii=imax,icnt
ccv          IC=ILa(ii)
ccv          DB(IR,IC)=1.0
ccv          db(ic,ir)=1.0
      db(1,ir)=1
   50   CONTINUE
c
c--use the counting matrix to define a nonredundant
c--list of the non-zero distance matrix entries
c--resulting from the search path
c
ccv         JCNT=1
      jcnt=0
         DO 30 J=1,natom
ccv         IF ( JCNT .GT. 1) GO TO 333
ccv         KK=J+1
ccv         DO 30 K=KK,natom
ccv         IF (DB(J,K) .EQ. 0.0) GO TO 30
ccv         IF (JCNT .EQ. 1) THEN
ccv         ILC(1)=J
ccv         ENDIF
      if (db(1,j) .eq. 1) then
         JCNT=JCNT+1
         ILc(JCNT)=j
      endif
   30   CONTINUE
  333         kcnt=jcnt
c
c--now write out list of atom numbers associated
c--with this path through the distance matrix
c
c
c--if ip=0 then have used this routine to condense
c--the list of connected points but do not wish
c--to print them
c
```

```
c--if ip=2 then print several forms of output
c
c
        IF (IP .GT. 0) THEN
c
C
C         sum=0.0
C         do 555 kq=1,kcnt
C         kqq=ilc(kq)
cxx
C         sum=sum+dmetric(ires,kres,kqq)
C 555     continue
C         xkcnt=kcnt
C         sum=sum/xkcnt
cc         AVGMET(MQ)=SUM
cc 665     continue
CCJCH        if (kcnt .eq. 1) go to 360
ccout           WRITE(6,243)IFAM,sum
cc           WRITE(6,743)(AVGMET(IH),IH=1,NPAIR)
 243       FORMAT(/'  DOMAIN',I5,'  avg value metric: ',f10.2)
cc 743       FORMAT( '  AVG METRICS: ',10F10.2)
ccout        WRITE(IW,909)KCNT
ccout 909    FORMAT('    SIZE',I5)
c
c--here i check for redundant info
c
 566       KEND=0
c
c--here we condense
c
      if (irowlist .ne. 0) then
      do 878 ijg=1,irowlist
      do 877 kjg=1,kcnt
      if (ilister(ijg,kjg) .ne. ilc(kjg)) go to 878
 877  continue
cccc  write(4,433)ijg,kcnt
c
      go to 766
 878  continue
      endif
 977  irowlist=irowlist+1
      NCLICKS=NCLICKS+1
      do 616 ijg=1,99
      ilister(irowlist,ijg)=0
      ICLICKS(NCLICKS,IJG)=0
 616  continue
C
      do 677 ijg=1,kcnt
C     write(6,*) kcnt,ijg
      JCK=ILC(IJG)
      ilister(irowlist,ijg)=ilc(ijg)
      ICLICKS(NCLICKS,JCK)=1
 677  continue
c
        DO 59 KKK=1,KCNT,10
        KST=KEND+1
        KEND=KEND+10
        IF (KEND .GT. KCNT) THEN
        KEND=KCNT
        ENDIF
ccout        WRITE(IW,404)(ILC(K),K=KST,KEND)
        if (kst .eq. 1) then
c       ccccccccccccccccccc original format ccccccccccccccccccccc
```

```
c        WRITE(4,403)(ires,kres,sum,(indice(ilc(k)),K=KST,KEND))
  403    FORMAT(2I4,'(',f6.3,'):',16(i4)/10(16x,16(i4)))
c       ccccccccccccccccccc my format ccccccccccccccccccccccccc
c     WRITE(41,4031)((indice(ilc(k)),K=KST,Kcnt))
      if(kcnt.eq.Ngroupe) then
          WRITE(41,4031)((indice(ilc(k)),K=KST,Kcnt))
          ncliques = ncliques +1
      endif
c         write (41,4031) (kcnt-kst+1)
c            WRITE(41,4031)((indice(ilc(k)),K=KST,Kcnt))
 4031   FORMAT(60 i2)
 4039   FORMAT(i5)
        go to 59
        endif
c       ccccccccccccccccc  original format ccccccccccccccccccccc
        WRITE(4,433)(indice(ilc(k)),K=KST,KEND)
  433   format(17x,16(i4))
c       ccccccccccccccccc    my format   ccccccccccccccccccccccccc
c        write (41,4331) (kend-kst+1)
c     WRITE(41,4331)((indice(ilc(k)),K=KST,KEND))
c 4331   format(40i2)
   59   CONTINUE
        ENDIF
  360   continue
  402   FORMAT(/)
  400   FORMAT(' (',F6.2,')',10I5)
  404   FORMAT('   CONFS',10I5)
  401   FORMAT(3X,10(1X,A4))
c
C--HERE I WILL WRITE OUT THE LIST OF ATOMS ASSOCIATED WITH
C--THE TEMPLATES INDEXED BY ILC VECTOR-- I.E. LEVEL ONE, TWO
C--AND A COMBINATION "LEVTHREE"
C
 766  continue
         DO 60 K=1,KCNT
         IR=ILC(K)
          IDMAT(IFAM,IR)=1

          DO 60 KK=1,KCNT
         IC=ILC(KK)
          dstart(ir,ic)=0.0
          dstart(ic,ir)=0.0
   60    CONTINUE
  100   format(/2i5,f10.5)
  300   Format(2i5,f10.5,5x,a4,1x,a4)
  305   FORMAT(A4,2I5)
        END
c
      subroutine reset(icnt)
CX
cxNEWCLIQUE 2/28/90
CX
      common/a/ila(8000),ilb(8000),dist(99,99)
       do 100 i=1,icnt
       ila(i)=0
       ilb(i)=0
  100   continue
        end
```

Clique1.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define true 1
#define false 0


/* Constantes du probleme */
int *mat, *column, *igroup, N, S, *ifingpe;

/* variables globales */
int x, k, *cliext;
int ncliques = 0;
FILE *result, *fopen();

int existcandext(int *candext, int *candidat, int *j);
void findcli(int *candidat);
void init_all(void);
void memoriserclique(void);
int read_igroup(void);
int read_matrix(void);
void reset_matrix(void);
void trt_clique(int *candidat, int *j);
void traitersommet(int *j, int *candidat);
void view_matrix(void);

int existcandext(int *candext, int *candidat, int *j)
{
  int y, z, ok;

  z = x + 1;
  ok = false;
  y = ifingpe[z] + 1;
  if (y == N) {
    return(true);
  }
  else {
    while (1) {
      if ((y != N) && (y != ifingpe[z+1] + 1)) {
      candext[y] = mat[*j * N + y] * candidat[y];
      ok = (ok || candext[y]);
      y++;
      }
      else {
      if (ok == false) {
        return(false);
      }
      else {
        if (y == N) {
          return(true);
        }
        else {
          ok = false;
          z++;
        }
      }
      }
```

```
    }
  }
}


void findcli(int *candidat)
{
  int j;

  j = ifingpe[x] + 1;
  while ((j != N) && (j != ifingpe[x + 1] + 1)) {
    traitersommet(&j, candidat);
  }
}



void init_all(void)
{
  int i;

  ifingpe = (int *) malloc (sizeof(int) * S);
  cliext = (int *) malloc (sizeof(int) * (S - 1));
  x = 1;
  k = 0;
  ifingpe[0] = -1;
  for (i = 1 ; i <= S ;i++)  {
    ifingpe[i] = igroup[i-1] + ifingpe[i-1];
  }
}


void memoriserclique(void)
{
  int z;
  char str[301], buf[4];

  ncliques++;
  str[0]='\0';
/*  sprintf(str, "\nC%02d is ", ncliques);*/
  for (z = 0 ; z < S ; z++) {
    sprintf(buf, "%2d ", cliext[z]);
    strcat(str, buf);
  }
/*  strcat(str, " --> ");
  for (z = 0 ; z < S ; z++) {
    sprintf(buf, "%2d ", column[cliext[z]]);
    strcat(str, buf);
  }*/
/*  strcat(str,"\n");*/
  fwrite(str, strlen(str), 1, result);
/*  printf("%s", str);*/
}


void traitersommet(int *j, int *candidat)
{
  int *candext;
```

```
  if (candidat[*j] == 1) {
    candext = (int *) malloc (sizeof(int) * N);
    if (existcandext(candext, candidat, j)) {
      cliext[k] = *j;
      if (k == S - 1) {
      memoriserclique();
      }
      else {
      x++;
      k++;
      findcli(candext);
      x--;
      k--;
      }
    }
    free(candext);
  }
  *j = *j + 1;
}



void trt_clique(int *candidat, int *j)
{
  int z;

  cliext[k] = *j;
  k++;
  for (z = ifingpe[x] +1 ; z < N ;z++) {
    candidat[z] = mat[*j * N + z];
  }
}



int read_igroup(void)
{
  int i;
  char filename[25], buf[11];
  FILE *indice, *fopen();

/*  printf("Entrer le nom du fichier d'indices -> "); */
/*  scanf("%s", filename); */
  strcpy(filename, "indice.dat");
  indice = fopen(filename, "r");
  if (indice != NULL) {
    fread(buf, 4, 1, indice);      /* Nombre d'indices */
    buf[4]='\0';
    S = atoi(buf);
    igroup = (int *) malloc (sizeof(int) * S);
    for (i = 0 ; i < S ; i++) {
      fread(buf, 5, 1, indice);
      buf[5]='\0';
      igroup[i] = atoi(buf);      /* Indices des colonnes */
    }
    fclose(indice);
    return 0;
  }
  else {
    return 1;
  }
```

```
}


int read_matrix(void)
{
  int i, j;
  char filename[25], buf[14];
  FILE *data, *fopen();

/*  printf("Entrer le nom du fichier contenant la matrice -> ");*/
/*  scanf("%s", filename);*/
  strcpy(filename, "mat.dat");
  data = fopen(filename, "r");
  if (data != NULL) {
    fread(buf, 12, 1, data);      /* 10 blancs */
    buf[12]='\0';
    N = atoi(buf);
    fread(buf, 3, 1, data);       /* 2 blancs + 0D + 0A */
    column = (int *) malloc (sizeof(int) * N);
    for (i = 0 ; i < N ; i++) {
      fread(buf, 2, 1, data);
      buf[2]='\0';
      column[i] = atoi(buf);      /* Indices des colonnes */
    }
    mat = (int *) malloc (sizeof(int) * N * N);
/*    reset_matrix();*/
    for (i = 0 ; i < N ; i++) {
      fread(buf, 3, 1, data);      /* 2 blancs + 0D + 0A */
      for (j = 0 ; j < N ; j++) {
      fread(buf, 2, 1, data);  /* Valeurs des ÇlÇments de matrice */
      buf[1] = '\0';
      mat[i * N + j] = atoi(buf);
      }
    }
    fclose(data);
    return 0;
  }
  else {
    return 1;
  }
}



void reset_matrix()
{
  int i, j;

  for (i = 0 ; i < N ; i++) {
    for (j = 0 ; j < N ; j++) {
      mat[i*N+j] = 0;
    }
  }
}



void view_matrix(void)
{
```

```
  int i, j;

  printf("La matrice est de taille %d\n", N);
  for (i = 0 ; i < N ; i++) {
    printf("%-2d ", column[i]);
  }
  printf("\n");
  printf("Il y a %d groupes. Les matchs sont de taille ", S);
  for (i = 0 ; i < S ; i++) {
    printf("%-2d ", igroup[i]);
  }
  printf("\n");
  for (i = 0 ; i < N ; i++) {
    for (j = 0 ; j < N ; j++) {
      printf("%d  ", mat[i*N+j]);
    }
    printf("\n");
  }
}



void main()
{
  int *candidat, j;
  char str[301];
  if (!read_matrix()) {
    if (!read_igroup()) {
/*      view_matrix(); */
      result = fopen("RESULTAT.DAT", "w+b");
      j = 0;
      init_all();
      candidat = (int *) malloc (sizeof(int) * N);
      while (j != ifingpe[x]+1) {
      trt_clique(candidat, &j);
      findcli(candidat);
      k--;
      j++;
      }
      sprintf(str, "\n Nombre de cliques pour Clique = %02d. ", ncliques);
      fwrite(str, strlen(str), 1, result);
      fclose(result);
      free(candidat);
      free(igroup);
      free(ifingpe);
      free(cliext);
    }
    free(mat);
    free(column);
  }
}
```

Proglit.for

```
C*****************************************************************
C*Programme de la litterature modifie avec lecture de densite
C*
C*****************************************************************
      parameter (MAXN=1000, MAXM=1000)
      INTEGER*2 N, ROW, COL, NGROUP, NCONF
      LOGICAL*1 MAT(MAXN, MAXN), TEMPLOG
      INTEGER EDGES, INDICE(99)
      REAL DENS, DIST(99,99)
      REAL*8 SEED
      INTEGER*2 ACTNODE(MAXN), TEMP
      INTEGER*2 ADJ(MAXM, MAXN), SOL(99,99)
      INTEGER*2 NODE, MIN, MINNODE, EDGE(MAXN)
      INTEGER*2 START(MAXN), LAST(MAXN), D, DTEMP, MAXCLIQUE, BEST(MAXN)
      REAL T1, T2, SORTTIME, TRAVTIME
      REAL TOTTIME

      OPEN (UNIT = 45, FILE = 'PROGLIT2.DAT', ACCESS = 'SEQUENTIAL',
     1      STATUS = 'NEW')
      OPEN (UNIT = 40, FILE = 'MAT.DAT', ACCESS = 'SEQUENTIAL',
     1      STATUS = 'OLD')
      OPEN (UNIT = 42, FILE = 'INDICE.DAT', ACCESS = 'SEQUENTIAL',
     1      STATUS = 'OLD')
      OPEN (UNIT = 43, FILE = 'DENSITE.DAT', ACCESS = 'SEQUENTIAL',
     1      STATUS = 'OLD')


      EDGES = 0
      READ (42, *) NGROUP
      READ (40, *) NCONF
      READ (43, 100) DENS
  100   FORMAT (31x,F4.2)
      N = NCONF
      MAXCLIQUE = 1
      READ (40, 432) (INDICE(J), J = 1, NCONF)
      DO 140 I = 1, NCONF
          DO 141 J = 1, NCONF
            DIST(I, J) = 0.0
141       CONTINUE
140   CONTINUE
      DO 143 I = 1, NCONF
          READ (40, 431) (DIST(I, J), J = 1, NCONF)
143   CONTINUE
      DO 144 I = 1, NCONF
          DO 145 J = 1, NCONF
            IF (DIST(I, J).GE. 1.0) THEN
                MAT(I,J) = .TRUE.
                EDGES = EDGES + 1
            ELSE
                MAT(I,J) = .FALSE.
            ENDIF
145       CONTINUE
144   CONTINUE

      DO 20 ROW =1,N
            MAT(ROW, ROW) = .FALSE.
20    CONTINUE
```

```
C     WRITE(*,*) ' '
C     WRITE(*,85) 'TOTAL EDGES', EDGES

C MAINTAIN POINTERS TO ORIGINAL MATRIX
      II = 1
      DO 25 NODE = 1,N
          ACTNODE(NODE) = NODE
25    CONTINUE

C ORDER NODES BY INCREASING EDGE DENSITY
C     WRITE (*,*)'DONNEZ LA DENSITE'
C     READ (*,*) DENS
      IF (DENS.GE..40) THEN
          DO 30 ROW =1,N
            EDGE(ROW) = 0
            DO 35 COL = 1,N
                IF (MAT(ROW,COL)) EDGE(ROW) = EDGE(ROW) + 1
35          CONTINUE
30        CONTINUE
          DO 40 NODE = 1, N-2
            MIN = N
            DO 45 ROW = NODE, N
                IF (EDGE(ROW).LT.MIN) THEN
                  MIN = EDGE(ROW)
                  MINNODE = ROW
                ENDIF
45          CONTINUE
            EDGE(MINNODE) = EDGE(NODE)
            IF (NODE.NE.MINNODE) THEN
                TEMP = ACTNODE(NODE)
                ACTNODE(NODE) = ACTNODE(MINNODE)
                ACTNODE(MINNODE) = TEMP
                DO 50 ROW = 1,N
                  TEMPLOG = MAT(ROW,NODE)
                  MAT(ROW, NODE) = MAT(ROW, MINNODE)
                  MAT(ROW, MINNODE) = TEMPLOG
50              CONTINUE
                DO 55 COL = 1,N
                  TEMPLOG = MAT(NODE,COL)
                  MAT(NODE,COL) = MAT(MINNODE,COL)
                  MAT(MINNODE,COL) = TEMPLOG
55              CONTINUE
            ENDIF
            DO 60 COL = NODE, N
                IF (MAT(NODE,COL)) EDGE(COL) = EDGE(COL) - 1
60          CONTINUE
40        CONTINUE
      ENDIF


      D = 1
      START(1) = 0
      LAST(1)      = N
      DO 65 COL = 1,N
          ADJ(1,COL) = COL
65    CONTINUE

C MAIN ALGOITHM

70    START(D) = START(D) + 1

      IF ((D+LAST(D)-START(D)).GE.MAXCLIQUE) THEN
```

```
          DTEMP = D
          D = D + 1
          START(D) = 0
          LAST(D)= 0

C DETERMINE NODE FOR TTHE NEXT DEPTH

          DO 75 COL = (START(DTEMP)+1), LAST(DTEMP)
            IF (MAT(ADJ(DTEMP,START(DTEMP)),ADJ(DTEMP,COL))) THEN
                LAST(D) = LAST(D) + 1
                ADJ(D,LAST(D)) = ADJ(DTEMP,COL)
            ENDIF
75        CONTINUE

C IF THE NEXT DEPTH DOESN'T CONTAIN ANY NODES, SEE IF A NEW MAXCLIQUE HAS
C BEEN FOUND AND RETURN TO PREVIOUS DEPTH

          IF (LAST(D).EQ.0) THEN
            D = D - 1
            IF (D.GE.MAXCLIQUE) THEN
                MAXCLIQUE = D
                DO 80 COL  = 1,D
                  BEST(COL) = ADJ(COL, START(COL))
80              CONTINUE
            ENDIF
          ENDIF
      ELSE

C PRUNE, FURTHER EXPANSION WOULD NOT FIND A BETTER INCUMBENT

          D = D - 1
      ENDIF

C CONTINUE TRAVERSAL UNTIL THE DEPTH OF ZERO IS REACHED
      IF (D.EQ.NGROUP) THEN
          IF (ADJ(D, START(D)).NE. NGROUP) THEN
              DO 180 COL=1, D
              SOL(COL,II) = ADJ(COL, START(COL))
180             CONTINUE
              II = II + 1
          ENDIF
      ENDIF
      IF (D.GT.0) GOTO 70

C OUTPUT THE MAXIMUM CLIQUE FOUND IN GRAPH

C     WRITE(*,85) ' MAXCLIQUE SIZE', MAXCLIQUE
      WRITE(45,*) 'NOMBRE DE SOLUTION', II - 1
      WRITE(*,*) 'NOMBRE DE SOLUTION', II - 1
C     WRITE(*,*) ' '
      DO 190 K = 1, II - 1
C         WRITE(*,*)  (ACTNODE(SOL(I, K)), I = 1,NGROUP)
190   CONTINUE
      DO 191 K = 1, II - 1
          WRITE(45,*)  (ACTNODE(SOL(I, K)), I = 1, NGROUP)
191   CONTINUE
C     WRITE(*,*) ' '
85    FORMAT(1X,A, I10)
90    FORMAT(1X,A, F10.4)
431   FORMAT(2X, <NCONF>F2.0)
432   FORMAT(2X, <NCONF>I2)
      STOP
      END
```

test.com

```
$ run rancli2
$ ty donnee.dat
$ run mixcli
$ run convers2
$ ty donnee2.dat
$ sortief
$ run densite
$ sortiet
$ ren sortie.dat densite.dat
$ sortief
$ @timing.com
$ sortiet
$ ty sortie.dat
$ ren sortie.dat testprog.dat
$ run Bip
$ sortief
$ search eric.dat nombre
$ search proglit1.dat nombre
$ search proglit2.dat nombre
$ search resultat.dat Nombre
$ sortiet
$ ty sortie.dat
$ ren sortie.dat ncl.dat
$ run bip
$ copy donnee.dat,donnee2.dat,testprog.dat,ncl.dat,densite.dat test.dat
$ ty test.dat
$ del donnee.dat;*,donnee2.dat;*,testprog.dat;* , ncl.dat;*
$ ty ncl.dat
$ run flo
```

timing.com

```
$ time1 = f$getjpi("","cputim")
$ run BRANDNEWCLI
$ time2 = f$getjpi("","cputim")
$ tcpub = time2 - time1
$ show sym tcpub
$!
$ time1 = f$getjpi("","cputim")
$ run FLO2
$ time2 = f$getjpi("","cputim")
$ tcpuf = time2 - time1
$ show sym tcpuf
$!
$ time1 = f$getjpi("","cputim")
$ run PROGLIT1
$ time2 = f$getjpi("","cputim")
$ tcpul1 = time2 - time1
$ show sym tcpul1
$!
$ time1 = f$getjpi("","cputim")
$ run PROGLIT2
$ time2 = f$getjpi("","cputim")
$ tcpul2 = time2 - time1
$ show sym tcpul2
$!
$ time1 = f$getjpi("","cputim")
$ run CLIQUE
$ time2 = f$getjpi("","cputim")
$ tcpuc = time2 - time1
$ show sym tcpuc
$ time1 = f$getjpi("","cputim")
$ run CLIQUE2
$ time2 = f$getjpi("","cputim")
$ tcpuc2 = time2 - time1
$ show sym tcpuc2
$!
$ time1 = f$getjpi("","cputim")
$ run CLIQUE3
$ time2 = f$getjpi("","cputim")
$ tcpuc3 = time2 - time1
$ show sym tcpuc3
```